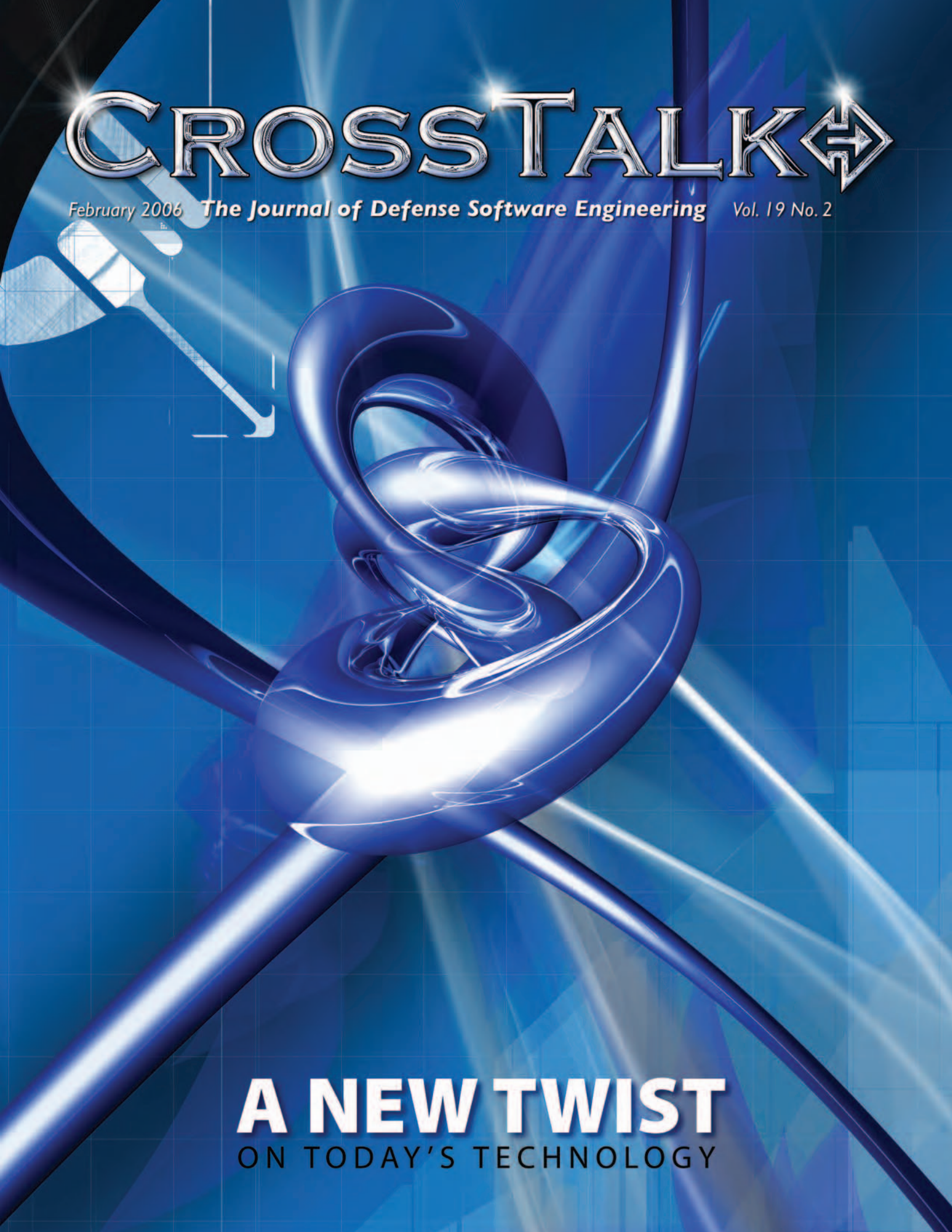
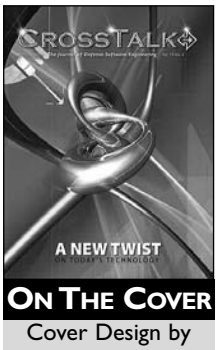

CROSSTALK
February 2006 The Journal of Defense Software Engineering Vol. 19 No. 2
A NEW TWIST
ON TODAY'S TECHNOLOGY

A New Twist on Today's Technology

Software Engineering Technology



ON THE COVER
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen. jensendesigns@aol.com

Departments

CROSSTALK

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Bob Zwitch*
DHS Co-Sponsor	*Joe Jarzombek*
NAVAIR Co-Sponsor	*Jeff Schwalb*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF), the U.S. Department of Homeland Security (DHS), and the U.S. Navy (USN). USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection. USN co-sponsor: Naval Air Systems Command (NAVAIR) Software Systems Support Center.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 22.

309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

What Is Up and Coming

If you have listened to a speech or presentation from our Department of Defense (DoD) leadership, then you have certainly heard the terms net-centric architecture and global information grid. The net-centric concepts are a key to the strategic vision for increasing our joint forces' capability. In an issue dedicated to "A New Twist on Today's Technology," it is important to consider our technology trends in the context of our warfighter's needed capability. The need for ubiquitous information and increased dependence on joint and allied cooperation should be a primary driving force of our technology innovations. The DoD vision will require more robust security, accurate/current data, reliable data transmission, improved wireless bandwidths, and overcoming other unforeseen technical challenges.

As you read this month's articles and consider your company's developments, think about how you are going to leverage information and global networks to give our warfighters the advantage.

Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor

Improving a Little at a Time

Many ideas to better our lives are not original inventions, but alterations of existing technology to create a new use. One example is the disposable cell phone with calling card capabilities. Customers buy the disposable cell phone with pre-set time on it, and then when the time has expired, dispose of it, recharge it for more time, or return it for a partial refund. The only thing new with this idea is the way existing technologies are leveraged for a new use. CROSSTALK's February theme authors focus on this strategy as they discuss new ways to take advantage of today's technologies.

We start this month with Dr. John A. Hamilton Jr.'s article, *DoDAF-Based Information Assurance Architectures*. The Department of Defense Architecture Framework (DoDAF) is not new, but the idea to leverage the DoDAF to develop information assurance architecture to enhance security is. Next, Dr. Kelvin Nilsen discusses a new specification for the Java programming language that will include safety-critical requirements in *Applying RAMS Principles to the Development of a Safety-Critical Java Specification*. Faced with the daunting tasks of illustrating the status of multiple programs with differing development schemes, Dr. Alistair Cockburn suggests a new way of combining information to provide this overall picture in *A Governance Model for Incremental, Concurrent, or Agile Projects*. In *Project Estimation With Use Case Points*, Roy K. Clemmons discusses how use cases can be used as more than just requirements documentation and also help with the estimation process. In our final online theme article, *Hard Skills Simulations: Tackling Defense Training Challenges Through Interactive 3D Solutions*, Josie Simpson suggests using computer simulations to augment hands-on training.

In our supporting article, three prominent members of the software estimation community discuss the similarities and differences of three popular software estimation models in *Software Estimating Models: Three Viewpoints*.

The CROSSTALK staff is constantly considering improvements in the way we provide information to our readers. The latest twist we are attempting is Really Simple Syndication (RSS) that will provide our online readers with an avenue to easily review the latest topics that have been published in each month's issue of CROSSTALK. Look for the RSS icon on our Web site soon.

Elizabeth Starrett
Associate Publisher

DoDAF-Based Information Assurance Architectures

Dr. John A. Hamilton Jr.
Auburn University

The Department of Defense (DoD) Architecture Framework (DoDAF) is the prescribed means for documenting information systems in the DoD and is an integral part of the Joint Capabilities Integration and Development System. The inclusion of DoDAF architectures in new system development is mandated in DoD acquisition regulations and is resource-intensive. Deriving information assurance architecture from DoDAF-compliant architecture is a relevant way to leverage the mandatory investment in DoDAF architectures. Every software engineer supporting the DoD should be aware of the increasing importance of information assurance and the need for holistic approaches to security. Information assurance architectures described in this article offer a verifiable holistic approach to security.

A logical extension of the Department of Defense (DoD) Architecture Framework (DoDAF) is to specify and describe information assurance architecture. Such architecture, while primarily built on the DoDAF systems views (SVs), can also be supported by technical standards views and validated by operational views (OVs). Defense software engineers need to be aware of the DoDAF, the mandated system architecture in the Defense Acquisition System [1].

The information assurance architecture diagrams are primarily derived from the system architecture. At the Department of Computer Science and Software Engineering at Auburn University, we use low-level architecture work products to document the major security components and the application mechanisms and their interrelationships. Successful information assurance strategies require holistic solutions, i.e., architectures are valid and internally consistent, so it is logical to leverage the mandated DoDAF architectures for the basis of information assurance architecture.

We look at information assurance architecture to support network analysis and design to mitigate distributed denial-of-service attacks on bandwidth. We document open ports and required services to support a systematic software vulnerability analysis. Finally, we use the OVs to perform a requirements analysis to validate our architecture. We translate operational requirements into a modified, but DoDAF-compliant Systems Interface Description (SV-1) product and a Systems Communications Description (SV-2) product. We then use those products as a basis for constructing the rest of the information assurance architecture.

> *"A simple and effective rule for security design is the principle of least privilege. That is, allow only the minimum essential connectivity and functionality."*

We then validate the information assurance architecture against the system requirements and verify it against security regulations and instructions. This article will describe how to construct DoDAF-compliant information assurance architecture based on the research efforts of Auburn University and the practical application of that research.

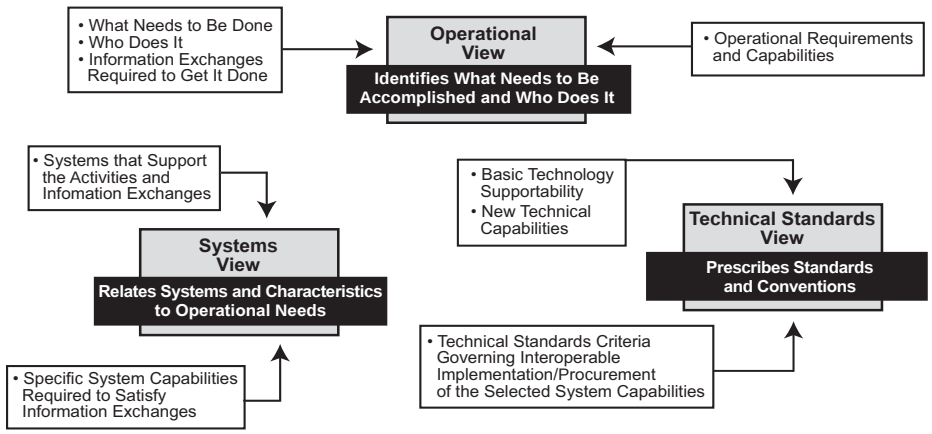
Overview of the DoDAF

This author is somewhat skeptical of mandates in general [2]. However, the requirement for DoDAF architecture is mandated by both the Defense Acquisition System and several chairmen. Joint Chiefs of Staff instructions, most notably the Joint Capabilities Integration and Development System [3], make it seem likely that the DoD soon will have a critical mass of DoDAF-compliant architectures. It makes sense to leverage those architectures beyond satisfying acquisition requirements.

The mandatory use of the DoDAF is prescribed in DoD Instruction 5000.2, in which the Joint Staff is the assigned proponent for OVs, while the under secretary of defense (Acquisition, Technology, and Logistics), leads the development of the SVs in collaboration with the services, agencies, and combatant commanders [1]. Volumes I and II of the DoDAF, plus the DoDAF Deskbook, total more than 1,500 pages of documentation; the description that follows is necessarily abbreviated.

As defined in the DoDAF [4], an OV is "a description of the tasks and activities, operational elements, and information exchange required to accomplish DoD missions." An SV is "a set of graphical and textual products that describes systems and interconnections providing for, or supporting, DoD functions. The SV associates systems resources to OV." These relationships are outlined in Figure 1.

Figure 1: *Relationship Between the DoDAF View*



Further discussion of these three views is available online in "Modeling Command and Control Interoperability: Cutting the

Gordian Knot" [5].

The technical standards view is essentially a listing of standards implemented by the systems in the architecture, and is now based on the DoD Information Technology Standards Registry found at <https://disronline.disa.mil>. Each communications system/network-enabled computer system defined in the SV will have an entry in the technical standards view outlining each standard used in the system.

The intellectual power of the DoDAF comes in the relationship between the OV and SVs. A tactical organization chart may be thought of as the starting point for an OV while a network connectivity diagram may be thought of as the starting point for an SV.

Simplistically, the OVs and SVs establish *what* systems must connect, and the SVs and technical standards view establish *how* systems must connect. From an engineering perspective, OVs are representations of requirements. The SVs describe how those requirements are implemented. DoDAF-compliant architectures constructed with this symmetry in mind have traceability between systems and requirements.

We exploit this traceability by considering the relationship of security policy to validating information assurance architecture. A simple and effective rule for security design is the principle of least privilege. That is, allow only the minimum essential connectivity and functionality. This is a principle easier to enunciate than it is to implement. Detailed requirements are needed to answer the question, "What is the minimum required functionality and connectivity?" From an information assurance perspective, security policy translates operational requirements into system requirements. This, then, is the basis of the methodology to develop information assurance architecture from DoDAF-compliant architecture.

Developing Information Assurance Architecture

What are the security requirements for the system(s) of interest? Our starting point is a set of OVs, specifically the OV-2 – the Operational Node Connectivity Description [6]. The DoDAF Deskbook [7] gives a high-level example of showing security attributes to a node as shown in Figure 2. Our methodology goes into more detail.

We start with the OV-2 Operational Node Connectivity Description. These nodes are the entities that are represented in the architecture. An object-oriented modeler would consider these nodes to be

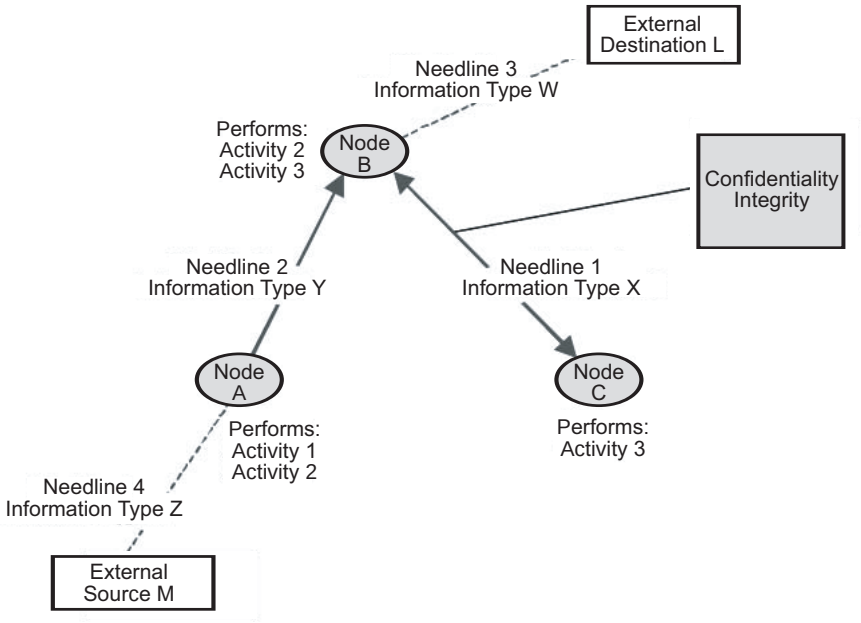


Figure 2: *DoDAF Deskbook Example of OV-2 With Security Attributes*

actors. An operational planner would consider these nodes to be a tactical element. These nodes can be represented at various levels of detail. A brigade combat team could be represented as one node, as a collection of battalion nodes, as an even larger number of company nodes, or quite probably a collection of nodes at different levels of abstraction.

For information assurance architecture, it is necessary to list every single system. A 99 percent solution is not effective! So, the cardinality may vary; that is, one node may represent one system, or one node may represent many systems. In our example, we use one-to-one mapping because it is easier. However, it is quite feasible, for example, to represent a network operations center as a single node and then have multiple system nodes that are part of the operations center node.

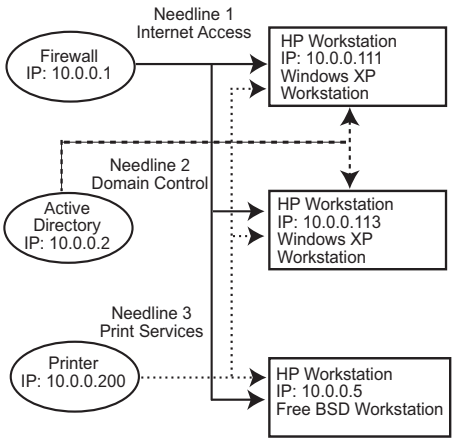
Consider the OV-2 Operational Node Connectivity Description of our Information Assurance Laboratory as shown in Figure 3.

Here we have gone beyond the minimal DoDAF standard and provided some additional information on what each node does. We then look at activities that each node is required to perform. The OV-5 Operational Activity Model [6] captures the required activities for each node. Now, it is a standard practice to go from the OVs to construct the SVs in new acquisitions. (In constructing *as-is architecture*, i.e., documenting existing systems, it is common to construct the SVs first.) To construct useful information assurance architecture, it is necessary to drill down to a greater level of detail. To achieve this, we base the security policy on the OVs. It could be argued that, in lieu of a separate security policy, the policy requirements be enumerated in an OV-6A Operational Rules Model [6] – we merely call it out separately from a DoDAF product for use as an operational security policy.

The construction of the security policy is oriented to the OV-2 and OV-5. For example, to allow file transfer protocol (FTP) access, we determine which nodes require activities that need FTP as opposed to some other, more secure transfer protocol. DoDAF traceability requires a consistent numbering policy. Our security policy representation is numerically indexed to the nodes and activities in the OVs. Since it is a DoDAF requirement that each system in the SVs (specifically the SV-1) be tied to a node in the OV-2, we now have a security crosswalk between the OVs (requirements) and the SVs (implementation).

Finally, for each system in SV-1, we list

Figure 3: *Simple OV-2 of the Auburn Information Assurance Laboratory*



Active Directory Controller

IP: 10.0.0.2
Description: Windows 2003 Server for running AD

connection to switch

PORT	STATE	SERVICE	VERSION
53/TCP	open	domain	Microsoft DNS
88/TCP	open	kerberos-sec	
135/TCP	open	msrpc	Microsoft Windows msrpc
139/TCP	open	netbios-ssn	
389/TCP	open	ldap	Microsoft LDAP server
445/TCP	open	microsoft-ds	Microsoft Windows 2003 microsoft-ds
464/TCP	open	kpasswd5	
593/TCP	open	http-rpc-epmap	
636/TCP	open	ldapssl	
1025/TCP	open	msrpc	Microsoft Windows msrpc
1026/TCP	open	msrpc	Microsoft Windows msrpc
1050/TCP	open	msrpc	Microsoft Windows msrpc
3268/TCP	open	ldap	Microsoft LDAP server
3269/TCP	open	globalcatLDAPssl	
8081/TCP	open	blackice-icecap	
Device type: general purpose			
Running: Microsoft Windows 2003/.NET			
OS Details: Microsoft Windows .NET Enterprise			

Figure 4: *Partial SV-1 With Information Assurance Details (Active Directory Controller)*

the minimal set of required services, processes, and open ports as shown in Figure 4, which shows a partial SV-1 derived from the OV-2 Active Directory Node in Figure 3. Based on the relationship between the OVs, security policy, and SVs, we now have a holistic information assurance architecture that can be verified on each system and validated against requirements.

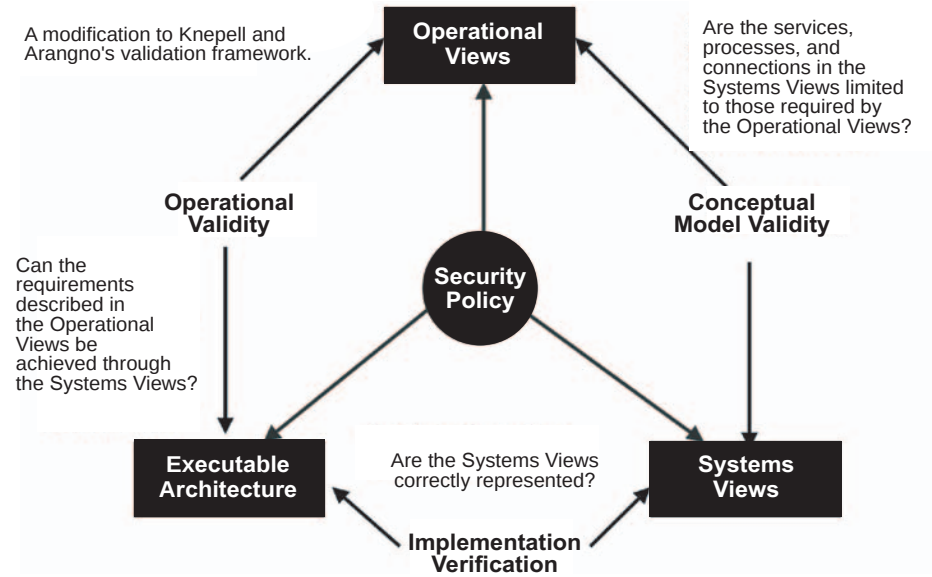
Verification and Validation of Information Assurance Architecture

Since the current Defense Acquisition System mandates using the DoDAF, developers have strong motivation to demonstrate that their architectures are valid and internally consistent. This holistic approach to system specification and connectivity is greatly useful in designing, verifying, and validating information assurance architecture. This relationship is shown in Figure 5.

OVs are fully defined in Volume 2 of the DoDAF [6]. Succinctly, OVs are representations of requirements. Consequently, there is a direct relationship between OVs and SVs. Figure 5 is a variation of Knepell and Arangno's validation structure adapted for information assurance application of the DoDAF [8].

An operational concept is not valid if it cannot be supported by the systems available in theater. So in this sense, the SVs validate the conceptual model of the OV as shown in Figure 5. Conversely, the validity of systems architecture can be evaluated against how well it supports the requirements documented in the operational architecture.

Figure 5: *Validating DoDAF-Based Information Assurance Architecture*



The employment of executable architecture adds a new and needed dimension to the verification and validation of DoDAF-compliant information assurance architecture. Executable architectures can assess the validity of an operational concept. While the SVs may provide the needed connectivity to support the operational concept described in the OVs, the SVs alone do not give sufficient insight into meeting operational performance and capacity needs. It can be argued that required performance can be extrapolated from the SVs, but executable architecture can provide a much more dynamic and flexible means of evaluation.

From an information assurance viewpoint, executable architectures can evaluate network and system design in terms of resistance and resiliency in the face of denial-of-service attacks [9]. A thorough discussion of executable architectures is beyond the scope of this article, but the heart of executable architecture is a network simulation constructed from the systems detailed in the DoDAF SVs and exercised by applying dynamic behavior across the system connections required in the OVs. Executable architectures are described in detail in [5].

Central to this validation structure is the security policy that is derived from the OVs, enforced in the SVs, and must be modeled in the executable architecture.

Conclusion

The DoDAF is mandated for use in the DoD Acquisition System – compliant architectures are a significant investment, and it makes sense to leverage this investment rather than undertaking a costly independent information assurance effort. We have briefly illustrated some specific ways to implement information assurance architecture and how to verify and validate such architecture.◆

References

1. Department of Defense. "DoD Instruction 5000.2." Operation of the Defense Acquisition System. Washington, D.C.: DoD, 12 May 2003: 2.
2. Hamilton Jr., J.A. "Why Programming Languages Matter." CROSSTALK Dec. 1997 <www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1997/12/languages.asp>.
3. Department of Defense. "Chairman,

Joint Chiefs of Staff Instruction 3170.01E." Joint Capabilities Integration and Development System. Washington, D.C.: DoD, 11 May 2005.
4. Department of Defense. DoD Architecture Framework Vers. 1.0 Vol. I: Definitions and Guidelines. Washington, D.C.: DoD, 9 Feb. 2004: 1-2 <www.defenselink.mil/nii/doc>.
5. Hamilton Jr., J.A. Modeling Command and Control Interoperability: Cutting the Gordian Knot. San Diego, CA: SCS Press, 2004: 85-99 <www.eng.auburn.edu/users/hamilton/security/spawar>.
6. Department of Defense. DoD Architecture Framework Vers. 1.0 Vol. II: Product Descriptions. Washington, D.C.: DoD, 9 Feb. 2004 <www.defenselink.mil/nii/doc>.
7. Department of Defense. DoD Architecture Framework Vers. 1.0 Deskbook. Washington, D.C.: DoD, 9 Feb. 2004: 2-79 <www.defenselink.mil/nii/doc>.
8. Knepell, P.L., and D.C. Arangno. Simulation Validation. Los Alamitos, CA: IEEE Computer Society Press, 1993.
9. Chatam, J.W. Using Strategic Firewall Placement to Mitigate the Effects of Distributed Denial of Service Attacks. Thesis. Auburn University, Aug. 2004.

About the Author

John A. "Drew" Hamilton Jr., Ph.D., is an associate professor of computer science and software engineering at Auburn University and director of Auburn University's Information Assurance Laboratory. Prior to his retirement from the U.S. Army, he served as the first director of the Joint Forces Program Office and on the staff and faculty of the U.S. Military Academy, as well as chief of the Ada Joint Program Office. He has a Bachelor of Arts in journalism from Texas Tech University, master's degrees in systems management from the University of Southern California and in computer science from Vanderbilt University, and a doctorate in computer science from Texas A&M University.

Auburn University
107 Dunstan Hall
Auburn, AL 36849
Phone: (334) 844-6360
Fax: (334) 844-6329
E-mail: hamilton@eng.auburn.edu

Web Sites

Air Force Research Laboratory

www.afrl.af.mil

The United States Air Force Research Laboratory (AFRL) leads the discovery, development, and integration of affordable warfighting technologies. The AFRL is a full-spectrum laboratory of approximately 9,500 people, responsible for planning and executing the Air Force's entire science and technology budget of nearly $1.7 billion, including basic research, applied research, and advanced technology development. AFRL partners include universities and industry with whom the AFRL invests almost 80 percent of its budget; customers include the Air Force major commands, which operate and maintain the Air Force's weapon systems.

National Aeronautics and Space Administration

www.nasa.gov

The National Aeronautics and Space Administration (NASA) conducts its work in four principle organizations, called mission directorates: aeronautics, exploration systems, science, and flight support. Closer to home, NASA's aeronautics team is working with other government organizations, universities, and industry to fundamentally improve the air transportation experience.

Practical Software and Systems Measurement Support Center

www.psmsc.com

The Practical Software and Systems Measurement (PSM) Support Center is sponsored by the Department of Defense (DoD) and the U.S. Army. It provides project managers with the objective information needed to successfully meet cost, schedule, and technical objectives on programs. PSM is based on actual measurement experience with DoD, government, and industry programs. The Web site also has the most current version of the *PSM Guidebook*.

Coming Events

March 6-9
Software Engineering Process Group (SEPG) 2006
Nashville, TN
www.sei.cmu.edu/sepg

March 13-15
International Symposium on Secure Software Engineering
Washington, D.C.
www.jmu.edu/iiia/issse

March 15-16
International Conference on Information Warfare and Security
Princess Anne, MD
http://academic-conferences.org/iciw/iciw2006/iciw06-home.htm

March 20-22
Association for Configuration and Data Management 11th Annual Technical and Training Conference
Sparks, NV
www.acdm.org/2006/conference.php

April 2-6
9th Communications and Networking Simulation Symposium
Huntsville, AL
www.scs.org/confernc/springsim/springsim06/cfp/cns06.htm

April 3-7
The 3rd International Conference on Software Process Improvement
Orlando, FL
www.icspi.com

April 10-12
3rd International Conference on Information Technology: New Generations
Las Vegas, NV
www.itng.info

May 1-4
2006 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

Applying RAMS Principles to the Development of a Safety-Critical Java Specification

Dr. Kelvin Nilsen
Aonix

Almost all software for aerospace and defense applications is required to satisfy reliable, available, maintainable, and safe (RAMS) objectives. While many RAMS issues are best addressed by requiring that software developers consistently adhere to particular development methodologies, a development team's selection of commercial off-the-shelf technologies, including choice of programming language, run-time environment, and libraries, may also impact the team's ability to satisfy RAMS requirements. This article evaluates a proposed specification for safety-critical Java in terms of RAMS principles, comparing the use of the draft safety-critical Java standard with traditional approaches based on C, and motivating the restrictions imposed by the safety-critical Java specification in comparison with use of traditional Java and the general purpose Real-Time Specification for Java. The RAMS solutions that have been designed for the proposed safety-critical Java specification apply equally well to a breadth of defense and aerospace application domains, including hard real-time mission-critical code for communication, sensing, guidance, and automation subsystems.

The Radio Technical Commission for Aeronautics (RTCA) DO-178B guidelines [1] are designed to span a range of criticality levels. The most life-critical software components in an avionics system are characterized as Level-A. Failure of a Level-A software component is considered *catastrophic*. Without this component, further flight and/or landing of an aircraft is considered impossible. Failure of a Level-C component is considered *major*, reducing the ability of a crew to cope with flight responsibilities, but not significantly increasing the risk of a crash. The safety-critical Java specification that is under development is designed to span the full range of DO-178B levels. In this regard, it addresses both life-critical and mission-critical systems.

Satisfying DO-178B certification requirements involves considerable engineering discipline. Enforcing this discipline is the responsibility of project managers and team leaders. Peer reviews are required at every step of the development process. Extensive documentation and accountability audit trails are required to ensure that no corners are cut in design, development, and testing of the safety-critical software. The DO-178B guidelines are independent of programming language choice. Regardless of programming language, engineers are required to address all of the same issues and gather all of the same documentation artifacts. You might reasonably ask, "What difference does the choice of programming language make?"

To answer this question, it is necessary to look more closely at some of the issues that must be addressed by developers of mission-critical and safety-critical systems. This article focuses on the programming language impact with respect to four broad issues: reliability, availability, maintainability, and safety (RAMS). In discussing these issues, we draw comparisons between using C, C++, the Real-Time Specification for Java (RTSJ) [2], traditional Java [3], and the proposed safety-critical specification for Java [4, 5]. Even though these issues can be addressed satisfactorily in a number of different languages, certain languages require less effort to address than others.

The draft safety-critical specification that is discussed here is currently under consideration for submission as a Java Community Process (JCP) standard. A prototype implementation of this specification is currently under development. Based on feedback from early evaluators, we expect to make appropriate refinements to the specification before submitting the final result for standardization under the JCP. We expect submission of the standard to take place during 2006. To track the progress of this ongoing specification and standardization work, refer to <http://research.aonix.com/jsc>.

Reliability

Among key considerations of developers focused on delivering high reliability are the following:

- The language and run-time environment must be sufficiently simple so they are easily understood.
- Ideally, the language and standard libraries behave consistently across platforms. Otherwise, programmers are likely to overlook incompatibilities when shifting their development efforts or migrating software components from one platform to the next.
- To help programmers manage complexity reliably, good programming languages support abstractions that allow programmers to separate concerns between independent components.
- Over the years, computer scientists have experimented with a variety of programming language features. Some very powerful features are easily misused, with far-reaching consequences. Certain features – such as implicit coercion of integer to Boolean – have been shown to be very error-prone. Programming languages that omit dangerous and error-prone features make it easier for developers to build reliable systems.

One of the main reasons Java has been such a popular alternative to C and C++ is because it is a much more portable programming language. This portability has resulted in a variety of important benefits. First, the standard libraries behave similarly across a wide variety of central processor unit (CPU) architectures and operating systems. Second, third-party developers of open-source and commercial off-the-shelf Java components are able to distribute these libraries in portable binary representations, without regard for on which platform they will be used. Third, developers of embedded systems can develop and test their software on fast, large-memory desktop machines, and then deploy the completed software on much slower, memory-limited embedded targets without concern that the code will behave differently in the embedded environment. Fourth, the *learning curve* for programmers consists only of learning the portable platform. No additional effort is required to learn the peculiarities of each implementation of the platform running on each different real-time operating system (RTOS).

These portability benefits are relevant to developers of safety-critical code. The

relevance of these benefits to system reliability is that (1) programmers are less likely to introduce errors because they have misunderstood or overlooked peculiarities of a particular Java implementation, and (2) the ability to reuse components across different platforms means a typical safety-critical deployment has a higher percentage of mature, time-proven software components in place versus custom-tailored software components that have never been used before.

Though the Java platform is portable with respect to functional behavior, standard-edition Java does not provide portability with respect to real-time issues. For example, the amount of memory allocated to create a particular data structure may vary significantly from one Java implementation to the next. And the scheduling of threads is also highly platform-dependent. To address these issues, the draft safety-critical Java specification carefully defines the precise semantics of a very small subset of the full Java Standard Edition libraries in combination with a small subset of the full RTSJ.

The selection of these libraries focuses on providing the minimal functionality required as a portable and extensible foundation upon which to build safety-critical systems. We annotate this set of libraries to characterize which services must be time- and memory-bounded, and we make these same annotations available to application developers so they can document their intentions with respect to the code they develop. A special safety-critical byte-code verifier (static analysis tool) enforces that method implementations are entirely consistent with the supplied programmer annotations. All of this clarifies which components can be reliably used in hard real-time contexts, including interrupt handlers. Further, determination of the memory and CPU time resources required for reliable operation of hard real-time software components is automatic.

Contrast this with the typical approach of a C or C++ developer. Since these languages were not designed for multi-threaded environments, the existing standards do not address the code generation issues that affect information sharing between threads. If a particular thread modifies a shared variable, even a variable that is defined as volatile, the propagation of the new value to other threads that are monitoring the same variable is highly nonportable. C and C++ programmers must understand the code generation model for each of the optimization modes they choose to use with their compiler. They must understand the underlying architecture's cache coherency model, and must study the underlying, real-time operating system's thread-scheduling semantics.

Often, the information required to develop reliable code is not well documented, and programmers have to spend considerable time and effort performing detective work to make sure they fully understand the platform they are targeting. This *detective work* often consists of trial-and-error experimentation. This may leave developers with lingering uncertainties as to whether their experimentation has uncovered all of the underlying platform's peculiarities and that they fully understand them. The software they write must be tested extensively to make sure it runs correctly on the targeted platform; however, the assumptions on which the correct operation of the software depends are rarely documented. If this software is ever moved to a different CPU, compiler, or RTOS, then extensive code review and retesting are required.

"Often, the information required to develop reliable code is not well documented, and programmers have to spend considerable time and effort performing detective work to make sure they fully understand the platform they are targeting."

The RTSJ exhibits some of the same difficulties encountered by C and C++ developers. Though this specification more carefully constrains real-time operation of threads than Standard Edition Java, it does not fully specify the semantics of the real-time libraries. Many of the capabilities offered within the RTSJ framework are optional, and the exact semantics of certain other features such as precisely when to trigger execution of a deadline-overrun handler are incompletely defined. This is one of the reasons that the draft safety-critical Java specification selects a subset of the full RTSJ framework. This subset specifically excludes capabilities that are difficult to define and implement in a portable way, and avoids many complex and costly features that are less relevant to developers of safety-critical or hard real-time systems. This results in a much smaller, more easily understood subset of core functionality, early implementations of which run more than three times faster than existing full RTSJ implementations on certain benchmarks.

Certain programming abstractions that are critical to developers of safety-critical code are totally irrelevant to typical developers of management information systems. Thus, languages like C, C++, and Standard Edition Java do not provide support for these abstractions. Consider, for example, the need of safety-critical developers to know the following:

1. Exactly how much real memory is required for the run-time stack of a safety-critical thread. (Note that safety-critical systems generally do not have hard disks and have no support for virtual memory or for dynamic expansion of a run-time stack.)
2. Exactly how much memory is required to represent a particular data structure that is going to be shared between multiple threads.
3. Exactly how much CPU time is required to reliably execute particular threads within real-time timing constraints [6].
4. Exactly how much time each thread might need to block while waiting for access to a shared resource that is required to complete a particular safety-critical task on schedule [6].

The safety-critical Java proposal addresses these issues by introducing standard meta-data annotations that allow programmers to constrain the behavior of particular methods. For example, an @StaticAnalyzable annotation denotes that the implementation of the method must adhere to particular style guidelines that allow the memory usage and the CPU time to be automatically determined. A special safety-critical byte-code verifier enforces that the code conforms to these style guidelines, and a separate static analysis tool determines the resource needs for each targeted platform.

Another special, hard real-time abstraction supported by the draft safety-critical Java proposal is a special synchronization mechanism known as priority ceiling emulation. With this mechanism, the programmer is required to specify an upper bound on the priorities of threads that might attempt to perform synchronized access to each lock. This upper bound is known as the ceiling priority.

When a thread acquires this lock, the thread's priority is immediately boosted to the ceiling priority. When the thread releases the lock, the thread's priority is restored to its original value.

Within the safety-critical profile, priority ceiling emulation is the only supported synchronization mechanism. A specialization of priority ceiling locks is known within the safety-critical profile as atomic locking. Programmers make use of a special syntax to identify objects that use atomic priority ceiling emulation to coordinate shared access between multiple threads. For each such object, the safety-critical byte-code verifier assures that the component does not perform any blocking operation while a given thread holds the object's atomic lock. With this byte-code enforcement in place, the implementation of atomic locks (on single-processor systems[1]) is very efficient and the worst-case blocking time to access an atomic lock is easily analyzed.

In particular, if a given thread is able to reach the point of entry to that lock, it is guaranteed that no other thread owns the lock. If another thread owned the lock, it would be executing at a higher priority so this thread would not be running. Thus, the blocking time is always zero. This important abstraction is recommended for all resource sharing among hard real-time safety-critical threads. It can only be implemented reliably through coordination between static analysis tools and run-time services. This coordination is provided within the safety-critical Java profile. It is not available in C, C++, Standard Edition Java, or the RTSJ.

Another important consideration is management of temporary scratch pad memory during the execution of hard, real-time, safety-critical threads. Temporary memory may be required to support digital signal processor analysis of sensor inputs, and to manage buffers for communication with redundant onboard, safety-critical modules and with remote systems that are, for example, providing air traffic control directives. The C malloc()/free() and C++ new()/delete() services are subject to memory fragmentation. Thus, they should not be used in memory-limited, safety-critical systems.

Java's automatic garbage collection system can defragment the dynamic memory heap. But the complexity and asynchrony of automatic garbage collection are very difficult to certify to the satisfaction of Federal Aviation Administration auditors. For this reason, the draft safety-critical Java profile provides an alternative memory management technique. We identify this approach as safe-scoped memory. It is a generalization of the RTSJ's scoped memory abstraction. In essence, the draft safety-critical profile allows objects to be allocated within the activation frames of each method.

In contrast with C and C++, which also allow objects to be allocated on the run-time stack, the safety-critical Java proposal uses programmer annotations to track the flow of stack-allocated objects; its byte-code verifier guarantees that no reference to a stack-allocated object lives longer than the object itself. This solves the all-too-common dangling pointer problem that plagues C and C++ development.

In contrast with the full-RTSJ scoped memory abstractions, which also prevent dangling pointers, the safety-critical profile detects scoped memory violations at compile time rather than at run-time. In summary, the proposed safety-critical Java temporary memory abstractions support much more reliable operations than common alternatives.

"Certain programming abstractions that are critical to developers of safety-critical code are totally irrelevant to typical developers of management information systems."

Availability

Availability addresses the requirement that high-integrity software must be always ready to perform its function. Availability is often measured in terms of a quantity of nines, representing the percentage of total time that the high-integrity system can be relied upon to fulfill its duty. For example, *seven nines* availability means the system is running reliably 99.99999 percent of the time. Note that seven nines operation allows only half a second of downtime per year.

Strategies for assuring high availability generally consist of a combination of the following:

- Take every reasonable action to maximize reliability as this will extend the *mean time between failures*. Reliability was discussed in the previous section.
- Minimize downtime whenever failures are encountered by doing the following:
  - Provide fast, deterministic restart of a failed system.
  - Provide fast, deterministic reconfiguration of software device drivers whenever failed hardware components might have to be replaced with upgraded hardware – if possible, upgrade device drivers and hot-swap hardware without rebooting.
- Support redundant computation and communication so that standby components can quickly take responsibility for ongoing services when particular components fail.

Providing a fast, deterministic restart of a failed system is an obvious requirement for high-availability applications. Achieving this objective is not trivial. Consider how long it takes to turn on typical computers and various smart gadgets such as cell phones. Startup is especially troublesome in typical Java environments, including compliant, full RTSJ implementations, because the startup process includes dynamic loading of byte code and translation of this byte code into native machine language by just-in-time (JIT) compilers.

The draft safety-critical Java profile addresses this concern by requiring static compilation, initialization, and linking of components. Unlike traditional Java, in which the initial values of many shared variables – even of so-called constant variables – depends on the order in which certain non-deterministic startup activities are performed, the safety-critical profile's byte-code verifier enforces fully deterministic initialization of shared static variables. The safety-critical Java linker binds all of the components together and initializes shared memory in the static load image. The large majority of this load image can be burned into read-only memory (ROM) and accessed directly out of ROM. Only objects with variable contents must be copied into random access memory at startup time.

Occasionally, highly available systems experience hardware failures. When hardware must be replaced, it is often necessary to replace software device drivers that control the hardware. Few real-time operating systems provide direct support for dynamic replacement of device drivers. Larger desktop operating systems usually support plug-and-play devices, but the protocols for using plug-and-play technologies are not especially reliable. Often, conflicts between device drivers supplied

by different vendors result in unreliable operation of the newly configured environment. A goal for the safety-critical Java profile is to support reliable and deterministic reconfiguration of device drivers, both for situations in which the device drivers are replaced without downtime and for situations in which hardware replacement requires system reboot.

As with many other issues, the safety-critical Java profile tackles this challenge using a combination of programmer annotations, special byte-code verification, and reliable run-time memory management services, specifically the following:

1. All of the memory consumed by a device driver is organized as a contiguous region of a budgeted size. If a particular device driver is removed, all of the memory previously set aside for that device driver is instantly reclaimed without any fragmentation issues. This memory can serve the needs of the replacement device driver.
2. The safety-critical Java profile provides annotations to allow programmers to describe the interface requirements of device drivers in sufficient detail to allow the static analysis tools to verify that a particular device driver is a suitable replacement for an existing device driver. Specifically, programmers can do the following:
   a. Characterize which inside/outside ports the device needs to read and write.
   b. Identify to which interrupt vectors the device driver needs to respond.
   c. Specify the maximum amount of time the device driver is allowed to hold particular interrupts disabled.
   d. Define the precise entry points whereby application code communicates with the device driver, and enforce that every invocation of these services is consistent with the interface expectations for that service.

Support for redundant computations and failover processing is not directly supported by the safety-critical Java profile. It is important to note that the Java platform was originally introduced as an Internet programming language. As such, there is considerable experience using Java for networked applications. Since the draft safety-critical Java profile establishes a strong foundation for reuse of portable, hard, real-time software components, it should be straightforward to develop portable libraries to support safety-critical, networked communications to support fault-tolerant and high-availability redundant computations.

Maintainability

With many safety- and mission-critical systems, fielded software must endure for many years, often several decades. During its useful lifetime, this software evolves in response to changing platform requirements, new communication protocols, integration of new functionality, and so forth. Over the lifetime of a particular system, it is common for the costs of software maintenance to far exceed the costs of the original software development. Typical maintenance activities include (1) fixing a bug, (2) addressing a performance issue, or (3) adding new functionality.

Maintaining real-time software is particularly difficult because the declared interfaces for C and C++ components do not reflect all the conditions required for reliable composition of real-time components. This means developers who are called upon to make changes to existing software cannot determine by looking at the component interface alone what rules they must follow for their changes to integrate reliably with other existing software. In particular, they do not know the following:

1. Whether incoming arguments might refer to temporary objects or permanent objects, and whether the referenced objects might be shared with other threads.
2. Whether memory resources have been budgeted to allow the implementation of a particular service to allocate permanent or temporary objects.
3. Which memory allocation budgets must be increased for this revised component to be able to reliably allocate additional memory.
4. Which memory allocation budgets can be decreased to make this component run more efficiently.
5. Which task cost-estimates must be modified if changes to this component affect its CPU time requirements.

"A goal for the safety-critical Java profile is to support reliable and deterministic reconfiguration of device drivers, both for situations in which the device drivers are replaced without downtime, and for situations in which hardware replacement requires system reboot."

Maintainers of real-time software must search for all the contexts in which particular components *reside* to determine what sort of changes they may make to those components without compromising system reliability.

Scalability is a generalization of maintainability. Many modern software systems experience evolutionary change that tracks Moore's Law [7]. As processors and computer memory decrease in cost and increase in capacity, software grows in size and complexity to match the new capacity. Studies of certain commercial, embedded software systems have observed that it is common for software size to double every 18 to 24 months [8].

Compared with C and C++, Java has shown tremendous strengths as a platform to support easy integration and economic scalability. This is because all of the Java software is very portable, and because strong object-oriented abstractions mean that independently developed components integrate cleanly, without compromising the integrity of each other's encapsulation boundaries. C, in contrast, offers very little to help manage the complexity of ever-expanding software systems. With its object-oriented features, C++ does much better than C at separating concerns of independent software development teams to facilitate software maintenance and scalability issues. However, the lack of true portability, the inherent complexity in the language itself, and its lack of automatic garbage collection makes C++ a more difficult tool than Java in its support for software maintenance and scalability.

The proposed safety-critical Java profile addresses these issues by doing the following:

1. Maintaining real-time software as *Vanilla Java* source code with Java Development Kit 5.0-style meta-data annotations to document the interface requirements associated with each software component.
2. Providing automatic consistency checking between independent interfaces, assuring that each method invocation satisfies the annotated interface requirements, that overriding method interfaces are consistent with the over-

ridden interfaces, and that method implementations are entirely consistent with the annotated method interface declarations.
3. Providing automated analysis to determine memory and CPU-time resource requirements to allow automatic configuration of resource budgeting and real-time scheduling each time any system component is modified.

Tools to automate the required consistency checking and resource needs analysis are not generally available for C and C++ development.

Safety

With respect to software systems, *safety* represents the notion that the computer system will do no harm. In this regard, safety is opposed to availability and reliability. The safest computer system might be the system that never gets powered on. Assuming that we are required to satisfy safety objectives in combination with reliability and availability objectives, the safety consideration consists primarily of satisfying safety certification requirements.

Of particular relevance to safety certification requirements is the mechanism for deployment of native machine code. Level-A certification requires that all code coverage analysis and testing be performed with the native machine language, and that responsibility for every machine code instruction and for every test case be traceable from original system requirements to architecture and design, to source code and test plans, and finally to machine code.

If certain machine instructions are not exercised sufficiently by the existing test cases, developers are required to analyze whether the code is really necessary to satisfy the system requirements. If that code is not necessary, the corresponding source code should be removed or restructured to make it more consistent with the system requirements. If the code is necessary, the test plan must be modified to make the test plan consistent with the original system requirements. In some cases, failure of test cases to cover all machine code reveals inaccuracies or inconsistencies in the original system requirements. In that situation, the original system requirements must be refined. Always, a complete traceability audit trail must be maintained.

Note that the traditional Java execution model is entirely inconsistent with this requirement for traceability from source code to machine code. Traditional Java virtual machines hide the translation of byte code to machine code within a JIT compiler that is part of the run-time environment. Some sophisticated virtual machines actually produce multiple native-code translations for each byte-code method, optimizing the code differently in each translation based on run-time profiling information. The safety-critical Java profile addresses this issue by supporting deterministic compilation, linking, and initialization model. The entire safety-critical application is translated to native machine code and linked into a ROM-loadable image prior to execution. Since this technology is designed to support safety-critical development, tools will facilitate mappings between machine code and the corresponding source code.

C and C++ development tool chains provide similar traceability support between source code and machine code.

Summary

Across the spectrum of RAMS objectives, the draft safety-critical Java specification offers important benefits over alternative approaches based on C, C++, traditional Java, or RTSJ Java. As commercial implementations of the proposed safety-critical Java standard become available, developers of safety-critical systems will be able to more economically deliver high-quality software that satisfies RAMS objectives.◆

References

1. Radio Technical Commission for Aeronautics, Inc. "RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification." Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
2. Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, Jan. 2000 <www.rtj.org>.
3. Sun Microsystems Inc. "The Java Language Environment: A White Paper." Mountain View, CA: Sun Microsystems, Inc., 1995 <http://java.sun.com/docs/white/langenv>.
4. Nilsen, K. "Making Effective Use of the Real-Time Specification for Java." San Diego, CA: Aonix, Oct. 2004 <http://research.aonix.com/jsc/rtsj.issues.9-04.pdf>.
5. Nilsen, K. "Draft Guidelines for Scalable Java Development of Real-Time Systems." San Diego, CA: Aonix, May 2005 <http://research.aonix.com/jsc/rtjava.guidelines.5-6-05.pdf>.
6. Klein, M., T. Ralya, B. Pollak, and R. Obenza. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Nov. 1993 <www.sei.cmu.edu/publications/books/other-books/rma.hndbk.html>.
7. Moore, Gordon. "Cramming More Components Onto Integrated Circuits." Electronics Magazine 19 Apr. 1965.
8. Bourgonjon, R. "The Evolution of Embedded Software in Consumer Products." International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, 1995 (unpublished keynote address).

Note

1. On a multiprocessor system, the implementation is a bit more complex, but the programming and static analysis abstractions are comparable. The safety-critical Java profile is designed to support straightforward migration of hard real-time code from single-processor to multi-processor implementations.

About the Author

Kelvin Nilsen, Ph.D., is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and commercial off-the-shelf safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. In 2003, Aonix acquired NewMonics. Nilsen has a Bachelor of Science in physics from Brigham Young University and a Master of Science and doctorate degree both in computer science from the University of Arizona.

Aonix
877 S Alvernon WAY
Tucson, AZ 85711
Phone: (520) 991-6727
Fax: (520) 323-9014
E-mail: kelvin@aonix.com

A Governance Model for Incremental, Concurrent, or Agile Projects

Dr. Alistair Cockburn
Humans and Technology

Use the model in this article to regain oversight of a complex multi-project that uses concurrent and incremental development strategies in differing amounts on different subprojects. The model makes this possible by estimating and then tracking intended versus actual functionality integrated either monthly or quarterly.

This article discusses a graphic that shows the status of larger projects, how to gather the information needed, read the status chart, and when needed, show more detailed breakdown for each subproject.

Motivation

Imagine a steering committee opening a large binder for a fairly large project, hoping to see its status. They are met by Gantt charts, cost expenditure tables, and if they are lucky, earned-value graphs. None of these quickly and accurately show the committee what is going on with the project, which components are in trouble, and where they stand with respect to delivery. Worse, many projects these days use incremental, concurrent, or agile approaches, which do not fit into the standard governance milestones of *requirements complete, design complete, code complete, and integration complete*.

This is not the article to review all the problems with that standard project governance model. The problems may, however, be briefly summarized as follows:

- The standard governance model is a single-pass waterfall model that has been roundly criticized over the years.
- It does not support incremental or concurrent approaches, both of which are being recommended more frequently these days.
- The Gantt chart does not show *percentage complete* very well, nor expected versus actual progress.
- The standard earned-value graph shows tasks complete well, but *tasks complete* does not reflect true progress on the project; true value in software development accrues suddenly (or fails suddenly) only at the moment at which the new features are integrated into the code base.

What is needed is a way that more accurately shows true value, more readily shows expected versus actual progress, and allows teams using different combinations of waterfall, incremental, concurrent, and agile approaches to show their varied strategies.

This article describes a proposed graphic that meets those constraints. Humans and Technology is starting to use it on a program that is fairly large for the private sector: about a dozen applications and two dozen supporting components and databases to be installed in various configurations in two dozen receiving sites. The graphic appears to work in our situation and has let us update a waterfall governance model to permit agile, concurrent, and incremental development.

The figures in this article are simplifications of the more complicated versions created for that program, so we have a sense that they scale reasonably. We see ways to update the graphic automatically, or even to update manually monthly for the steering committee meeting. While not fully tested, it shows promise.

Here are the details – please write me if you try it out and come up with improvements. (Note: The figures in this article are printed in black and white; the online version [1] uses color.)

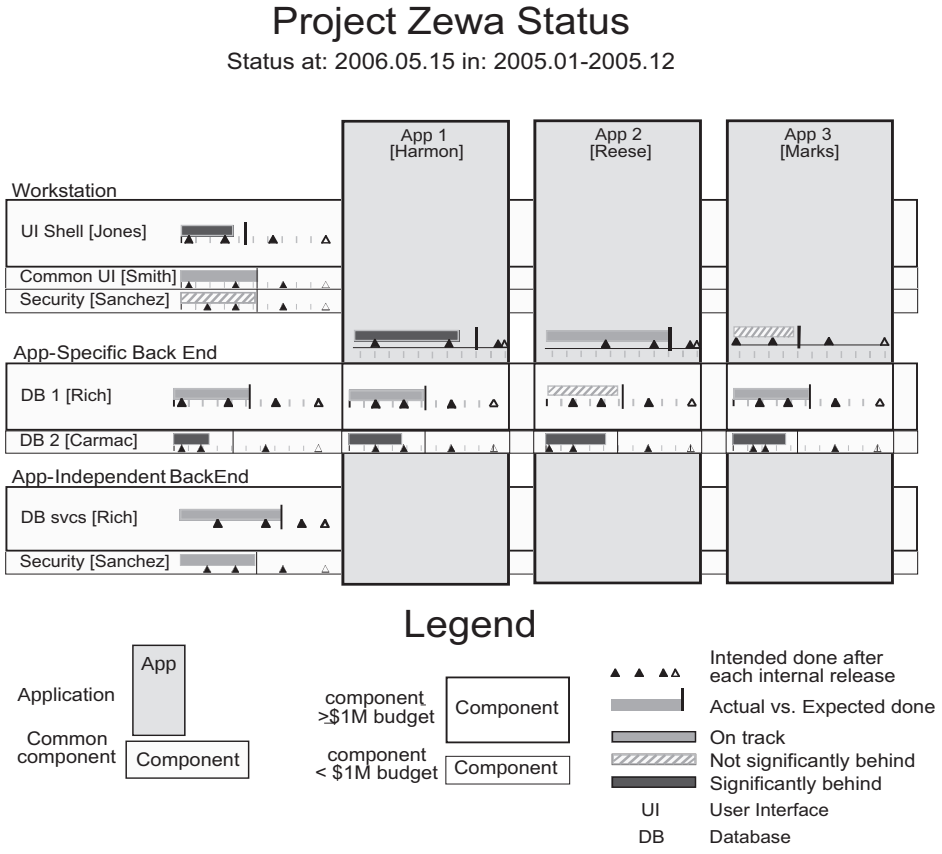
The Graphic

The graphic contains the following elements:

- The components being built, how each fits into the system architecture, and the person responsible for each.
- Codependency relations between horizontal and vertical components.
- The incremental growth strategy for each component over the project life.
- The expected versus actual amount of work completed to date.
- A color or shading to mark the alarm level for each component.

The following is a discussion of those topics in three categories: the system structure, the intended strategies, and

Figure 1: *Governance Graphic*



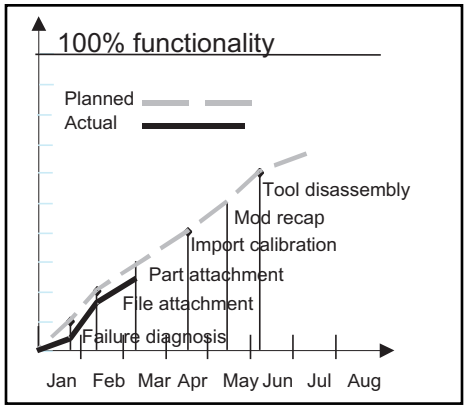


Figure 2: *Burn-Up Chart*

expected versus ideal progress.

System Structure

In Figure 1, the three vertical rectangles indicate *applications* – the items bought separately for end-user functionality. The seven horizontal rectangles indicate service components needed across applications – on the user's desktop or the back end. Two of the horizontal components cross *in front of* the applications to indicate that the horizontal component contains a specific, different functionality or data for each application. Domain databases that get extended for each new application are likely to be among these application-specific, back-end components (more on this later).

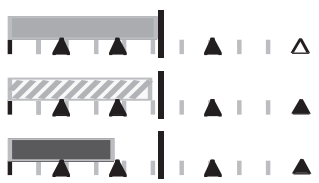
The system shown in Figure 1 is fairly simple. The first project for which I drew this graphic had 17 horizontal and 15 vertical components, and additional coloring to show legacy components that were to be removed over time. We were still able to draw it legibly on legal-sized paper.

Intended Strategies

Although incremental development has been around much longer than the agile movement, the question of how to show different incremental strategies for governance purposes – preferably in a small space – is still open.

Figure 3: *Target Progress Markers*

Figure 4: *Intention-Completion Bars*



Agile project teams measure progress not according to how many requirements have been gathered, but by how much running functionality has been designed, programmed, and integrated (Ron Jeffries neatly calls these running tested features, or RTF [2]). A common way to show the growth of RTF is through burn-up charts, as in Figure 2, which shows the expected versus actual integration of a set of workflow components by month.

Agile burn-up charts are very similar to traditional earned-value charts with one crucial exception: The team only gets credit when the features are integrated into the full code base and the result passes testing [3]. This single chart shift makes a big difference in the reliability of the progress data presented.

Burn charts show more than we need and take too much space for governance oversight purposes. To reduce their size and information, we use the idea of an *internal release* (IR).

A team that cannot deploy its system to live users every few months can *pretend* to deploy the system. It can test, integrate, and deploy the system to the computer of one or two *friendly* but real users. The team thus exercises the end of their development process and gets true user feedback. Putting the system in front of real users (as opposed to a test machine in the basement) motivates the development team to take their work seriously.

Such an IR should happen every one, two, or three months. There are many reasons not to deploy fully every three months, but there is almost no reason not to carry out an IR. These IRs fit neatly into a monthly or quarterly reporting mechanism.

With RTFs and IRs, we are ready to capture various development strategy or strategies that might show up on an incremental development project.

In Figure 3, the vertical tick-marks show 10 percent units of completed RTF from left to right (100 percent complete at the right). The triangle milestone markers show the amount of RTF the team targets to have completed and integrated at each IR milestone. Figure 3 shows three teams' strategies as follows:

- The top team plans to get less than 10 percent of its functionality in place in the first IR, and to add functionality in roughly equal thirds after that.
- The middle team intends to get 25 percent done in the first quarter, 60 percent by the end of the second quarter, and almost 85 percent through the third quarter (possibly so they have time to fix mistakes in the fourth quarter).
- The bottom team expects to get almost 20 percent completed and integrated in each of the first two quarters, and then to speed up and get 30 percent done in each of the last two quarters.

Alert readers will notice that these tickmark drawings capture the vertical axis of the burn-up charts at the IR times.

These small diagrams let different teams work in different ways and report on their intentions. This is our goal.

Expected Versus Ideal Progress

Intention-Completion Bars

Figure 4 adds to Figure 3 the work actually completed compared to the work targeted for any point in time.

In Figure 4, the taller vertical bar moves from left to right within each IR period to show the current targeted accomplishment. It can run at a constant rate within each period according to the calendar, or it can be synchronized with the team's iteration plans (two- to six-week planning and tracking time windows). The shaded rectangles show the functionality (RTF) completed and integrated to date.

In Figure 4, we see that the top team is delivering according to schedule, the middle team is a little behind, and the bottom team still has not finished the work scheduled for the second IR.

Two comments must be made at this point about RTF. The first is that not all final deliverables consist of *features* that *run*. Content databases and end-user documentation are examples. Teams can create intention-completion bars for whatever their final deliverables are, since those bars show growth of accomplishment over time.

The second comment is that measures not tied to RTF are naturally hazardous since it is so easy to start tracking completion of artifacts that do not directly get bought by customers. Linking accomplishments to RTF makes the reporting of actual *value* both easier and more accurate.

Application-Specific Components

Horizontal components such as application databases require new work and new content for each new application. Progress on these application-specific horizontal components is typically difficult to report on since *where they are* varies from application to application.

To show the status of such a component, we use intention-completion bars for the independent portion of the component and for each application it must serve. This lets the teams move at different speeds as suits their particular situations, and allows the steering committee to see each team's status.

Summarizing the Graphic

Let us review the elements of the graphic briefly:

- The rectangles represent components, subsystems, or applications. Vertical rectangles show applications; horizontal ones show components that get used across multiple applications (this could be reversed for better layout if, for example, there are many applications and only a few cross application components).
- Each rectangle shows the place of the component in the overall architecture: The top set of horizontal components reside on the desktop, the middle and bottom sets of horizontal components reside as back-end services. The horizontal rectangles running *behind* the applications get created independently of the applications; the horizontal rectangles running *in front of* the applications require application-specific work or content.
- Intention-completion markers are created for each component. They show the percentage of RTF intended for completion at each IR milestone, the expected and the actual current accomplishment, and the alarm level. Intention-completion bars are created for each component and for each intersection of application-dependent components.

Collecting the Information

The information rendered in Figure 1 also fits into a spreadsheet, a more useful place to keep it while gathering and updating the information. We can use automated tools to gather information about each component every week or two, and roll up each team's accomplishments into reports at various levels. The highest level is the one that gets *painted* onto the graphic either by hand or automatically. (The graphic can be generated automatically using graphic markup languages, but that programming effort may take longer than simply coloring the bars each month).

Gathering the Estimates

It is one thing to say, "We intend to be 20 percent done after the first internal release," but the steering committee needs to know, "Twenty percent of what?" Being behind on 20 percent of two use cases is very different than being behind on 20 percent of 80 use cases.

Project/ Component	Sub-Component	Owner	Percent Done in IR1				Total Size	Units	Confidence in Estimate (L, M, H)
			IR 1	IR 2	IR 3	IR 4			
Desktop	frame	Mr. A	0	20	80	100	30	UI widgets	Med :-|
Desktop	APIs	Mr. A	20	50	80	100	60	API calls	Lo :-(
App 1	UI	Mr. B	5	60	90	100	450	UC steps	Lo :-(
App 1	app	Mr. B	10	60	90	100	450	UC steps	Hi :-)
App 1	bus svcs	Mr. B	5	50	80	100	450	UC steps	Lo :-(
DB 1	setup	Ms. C						??	Lo :-(
DB 1	App 1	Ms. C					60	codes	Lo :-(
DB 1	App 2	Ms. C					10	codes	Lo :-(
DB 2	setup	Ms. C						??	Lo :-(
DB 2	App 1	Ms. C					2,000	entity attributes	Lo :-(
DB 2	App 2	Ms. C					1,500	entity attributes	Lo :-(

API - Application Program Interface, App - Application, UC - User Class

Table 1: *Estimating Spreadsheet*

> *"A team that cannot deploy its system to live users every few months can pretend to deploy the system. It can test, integrate, and deploy the system to the computer of one or two friendly but real users."*

To capture the *of what* for tracking, we need three pieces of information. The first, "What is the unit of accomplishment?" often consists of use cases, or more likely, individual steps in use cases. Sometimes something quite different is appropriate. A desktop component might have as units of accomplishment user interface (UI) widgets (frames, pull-down lists, buttons) and interface calls used by the applications. A database might have entities and attributes, a Web site might have articles and images, a medical database might have medical codes as a unit of accomplishment.

The second piece of information is, obviously, "About how many units do you expect to create?"

The third piece of information is the confidence level on the estimate. At the beginning of the project, it is appropriate to have low confidence ratings in the estimates: "We expect somewhere between 15 and 50 UI widgets, call it 30, plus or minus 50 percent;" however, that comes with the caution, "You called me into this room and made me give you numbers, but it's not like I have a really good basis for those numbers!"

The initial rough-size estimate is still useful for getting an early handle on the size and shape of the thing to be built. That is why the information is collected even when the confidence rating is low. Marking a low confidence rating is useful to the project leaders because they can then raise the priority of getting enough information to improve the confidence level.

Needless to say, the estimate should be updated at the start of successive iterations with raised expectations about its accuracy and confidence levels.

Table 1 shows a spreadsheet that can be used to capture the estimates. Note that the confidence rating is accompanied by a smiling, neutral, or frowning face to visually tag this important information.

Gathering the Status

To tag the timeline, we need to give each iteration or planning window a milestone number such as an IR completed then followed by iteration completed. Thus, milestone 0.3 means the end of the third iteration before the first IR, and milestone 2.1 means the end of the first iteration after the second IR.

After iterations, the teams send in

Project/ Component	Sub-Component	Owner	Percent Done In				Total Size	Units	Expected at IR 2.3 (percent)	Expected at IR 2.3 (units)	Actual at IR 2.3 (units)
			IR 1	IR 2	IR 3	IR 4					
Desktop	frame	Mr. A	0	20	80	100	30	UI widgets	50%	15	15
Desktop	APIs	Mr. A	20	50	80	100	60	API calls	65%	39	36
App 1	IU	Mr. B	5	60	90	100	450	UC steps	75%	337	310
App 1	app	Mr. B	10	60	90	100	450	UC steps	75%	337	320
App 1	bus svcs	Mr. B	5	50	80	100	450	UC steps	65%	292	280

Table 2: *Summary Spreadsheet*

their RTF numbers, which get rolled up into a summary spreadsheet at any level of granularity desired. The nice thing here is that this roll-up can be produced automatically with simple tools. Table 2 shows how the first few rows of such a spreadsheet might look after iteration 2.3.

The Status Report Packet

The graphic in Figure 1 serves as a good *summary page* of the package put in front of the steering committee. That package also needs detail pages for the separate subprojects.

Table 3 shows a sample detail page. This detail page has three sections after the header:

- A status/targeted/deferred and risk snapshot for each section of work within the component.
- A commentary, including surprises and lessons learned during the previous period.
- Cost roll-up information.

The most unusual part of this status page is the way in which the intention-completion bars are constructed to describe the strategy and accomplishments of non-RTF work.

Intention-Completion Bars for Non-RTF Work

When someone sees a component marked with a high-alarm status bar on the summary page, they will naturally want to read more detail. They will need to understand what is happening with respect to requirements gathering, UI design, design and programming, and user documentation.

The good news is that we can use the intention-completion bars to show progress within each specialty, whether the team is using a sequential (waterfall) strategy or a concurrent strategy. Figures 5 and 6 illustrate the two.

Figure 5: *A Sequential Development Strategy*

Figure 6: *A Concurrent Development Strategy*

Figure 5 shows a team planning to work in sequential fashion. They plan to finish all their requirements in the first period. They do not plan on starting either the UI design or the programming in that period. They expect to get the UI design fully complete in the second quarter. They plan to get perhaps 10 percent of the programming done in the second quarter, and the rest done equally in the third and fourth quarters.

Figure 6 shows a strong concurrent strategy. This team plans to get not quite a third of their requirements settled in the first period, and to have nearly as much UI design and programming done as requirements gathered. The requirements people will lead the UI design people by a small amount, and the UI design people will lead the programmers by a small amount, but otherwise these groups will run parallel to each other. They intend to continue in this fashion throughout the entire project.

In this article, I do not wish to indicate that either approach is superior to the other. What is important here is that both sequential and concurrent strategies (and many combinations) can be shown using the intention-completion bars.

Status, Targeted, Deferred, and Risks

For any component, the steering committee members will want to see the following at the top of the detail page:

- The intention-completion bars for the whole component from the summary sheet, and for the work efforts within the component, including non-RTF work as just described.
- What was targeted for accomplishment during this reporting period?
- What work is being deferred from this period into the next?
- The dominant problems each subteam is facing or the risks they expect.

The risks and problems column lets the team signal for help, whether that means more people, more equipment, more time with customers, etc.

Surprises, Lessons Learned, Items Needing Special Attention

The middle of the page allows the team to reflect and report on what happened during the reporting period.

The first section describes the surprises discovered. On projects I have visited, these have included the programmers not getting as much done as expected, a piece of technology not working as expected, or, conversely, a new practice such as daily stand-up meetings being effective.

The second section describes the lessons to be taken out of the period's work. These might include multiplying developer estimates by a factor before committing to them, doing technology spikes before committing to technology, or choosing to keep the new, daily, stand-up meetings. These must be truly lessons *learned* within the period, not speculations on what might work in the future.

The third section is for anything the team wishes to report on. It may expand on risks or highlight some particular worry to which they will be paying close attention.

Cost Roll-up

Finally, the steering group needs to see how fast the money is being used. This section may be presented in tabular or burn-up form, and include staffing sizes as well as budget information as desired.

Summary

The first contribution of this article is the description of the intention-completion graphic, showing the following:

- The strategy that the team has in mind for its work, whether sequential or concurrent.
- How much the team had expected to have done at this reporting point.
- How much the team actually has done at this point.

The intention-completion graphic is important because it allows different teams to choose different strategies and report on them, all in the same format. The absence of a common reporting format has been a painful point for incremental, concurrent, and agile projects for a long time.

The second contribution is the project summary graphic and its spreadsheet counterpart. The spreadsheet allows the leadership team to collect estimates and plans at a very early point in the project, and easily update these by using automated tools. The graphic provides a way to show at a glance the entirety of a quite complex project. This addresses the questions, "What are we building?" and "How are we doing?"

The third contribution is the description of a sample, one-page detail sheet (see Table 3) for each component or subproject. This page shows at a glance the strategies and status within the subproject, along with key information the steering committee needs to understand and respond to.

The resulting packet of information allows people who meet only once a month or quarter to assess the intentions and status of projects that use various mixtures of waterfall, incremental, concurrent, and agile strategies.

If you use this model and find ways of improving it, please let me know at <acockburn@aol.com.>◆

References

1. Cockburn, A. "A Governance Model for Incremental, Concurrent, or Agile Projects." CROSSTALK Feb. 2006 <www.stsc.hill.af.mil/crosstalk/2006/02/0602Cockburn.html>.
2. Jeffries, R. "A Metric Leading to Agility." XProgramming.com 14 June 2004 <www.xprogramming.com/xpmag/jatRtsMetric.htm>.
3. Cockburn, A. "Earned Value and Burn Charts." Technical Report. Humans and Technology, Apr. 2004 <http://alistair.cockburn.us/crystal/articles/evabc/earnedvalueandburncharts. htm>.

Detail Sheet for: UI Shell
Product Manager: *Jones*
Status at: *2006.05.15*

	Targeted Accomplishment	Work Being Deferred	Dominant Problem/Risk
Composite 	<What the accomplishment was to be in this period for this sub-project.>	<What got moved out of this period into the next period?>	<The dominant risk for this sub-project.>
Requirements 	<The amount of requirements intended to be completed in this period.>	<What requirements got moved out of this period into the next period?>	<The dominant risk for the requirements gathering effort.>
UI Design 	<The amount of user interface design intended to be completed in this period.>	<What user interface design got moved out of this period into the next period?>	<The dominant risk for the UI designers.>
Program 	<The amount of RTF intended to be integrated in this this period.>	<What programming got moved out of this period into the next period?>	<The dominant risk for the programmers.>
User Doc. 	<The amount of end-user documentation intended to be completed in this period.>	<What end-user documentation got moved out into the next period?>	<The dominant risk for user documentation.>

Surprises this Period:
<Surprises the manager or the team discovered (e.g., the productivity of the programmers wasn't as high as expected).>

Lessons Learned this Period:
<The lessons to be taken out of the period's work (e.g., in the future, multiply developer estimates by a factor of 1.5 before committing to them).>

Items Needing Special Attention:
<Anything the team wishes to report out. It may expand on risks, or highlight some particular worry.>

Cost/Budget

	This Period	Total to Date
Expected	$	$
Actual	$	$

Table 3: *Detail Sheet for UI Shell*

About the Author

Alistair Cockburn, Ph.D., is an internationally respected expert on object-oriented design, software development methodologies, use cases, and project management. The author of two Jolt Productivity award-winning books, "Agile Software Development" and "Writing Effective Use Cases," as well as the perennial favorite, "Surviving OO Projects," he was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group in 1992, served as special advisor to the Central Bank of Norway in 1998, and has worked in companies from Scandinavia to South Africa, North America to China. Internationally, he is known for his seminal work on methodologies and use cases, as well as his lively presentations and interactive workshops. Many of his materials are available online at <http://alistair.cockburn.us>.

Humans and Technology
1814 Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
E-mail: acockburn@aol.com

Project Estimation With Use Case Points

Roy K. Clemmons
Diversified Technical Services, Inc.

Software developers frequently rely on use cases to describe the business processes of object-oriented projects. Since use cases consist of the strategic goals and scenarios that provide value to a business domain, they can also provide insight into a project's complexity and required resources. This article provides an introduction to the Use Case Points method that employs a project's use cases to produce a reasonable estimate of a project's complexity and required man-hours.

Use case modeling is an accepted and widespread technique to capture the business processes and requirements of a software application project. Since use cases provide the functional scope of the project, analyzing their contents provides valuable insight into the effort and size needed to design and implement a project. In general, projects with large, complicated use cases take more effort to design and implement than small projects with less complicated use cases. Moreover, the time to complete the project is affected by the following:

- The number of steps to complete the use case.
- The number and complexity of the actors.
- The technical requirements of the use case such as concurrency, security, and performance.
- Various environmental factors such as the development teams' experience and knowledge.

An estimation method that took into account the above factors early in a project's life cycle, and produced an estimate within 20 percent of the actual completion time would be very helpful for project scheduling, cost, and resource allocation.

The Use Case Points (UCP) method provides the ability to estimate the man-hours a software project requires from its use cases. Based on work by Gustav Karner [1], the UCP method analyzes the use case actors, scenarios, and various technical and environmental factors and abstracts them into an equation. Readers familiar with Allan Albrecht's "Function Point Analysis" [2] will recognize its influence on UCP; function point analysis inspired UCP.

The UCP equation is composed of three variables:

1. Unadjusted Use Case Points (UUCP).
2. The Technical Complexity Factor (TCF).
3. The Environment Complexity Factor (ECF).

Each variable is defined and computed separately using weighted values, subjective values, and constraining constants. The weighted values and constraining constants were initially based on Albrecht, but subsequently modified by people at Objective Systems, LLC, based on their experience with *Objectory* – a methodology created by Ivar Jacobson for developing object-oriented applications [3]. The subjective values are determined by the development team based on their perception of the project's technical complexity and efficiency.

Additionally, when productivity is included as a coefficient that expresses time, the equation can be used to estimate the number of man-hours needed to complete a project. Here is the complete equation with a *Productivity Factor* (PF) included:

UCP = UUCP * TCF * ECF * PF

The necessary steps to generate the estimate based on the UCP method are the following:

1. Determine and compute the UUCPs.
2. Determine and compute the TCFs.
3. Determine and compute the ECFs.
4. Determine the PF.
5. Compute the estimated number of hours.

Sample Case Study

In the sections that follow, the UCP method is retroactively applied to a Web application developed by the author. This after-the-fact approach provides a practical way to establish a baseline PF for projects already completed. As described later, the PF helps determine the number of man-hours needed to complete the project.

UUCPs

UUCPs are computed based on two computations:

1. The *Unadjusted Use Case Weight (UUCW)* based on the total number of activities (or steps) contained in all the use case scenarios.
2. The *Unadjusted Actor Weight (UAW)* based on the combined complexity of all the actors in all the use cases.

UUCW

The UUCW is derived from the number of use cases in three categories: *simple*, *average*, and *complex* (see Table 1). Each use case is categorized by the number of steps its scenario contains, including alternative flows.

Keep in mind the number of steps in a scenario affects the estimate. A large number of steps in a use case scenario will bias the UUCW toward complexity and increase the UCPs. A small number of steps will bias the UUCW toward simplicity and decrease the UCPs. Sometimes, a

Table 1: *Use Case Categories*

Use Case Category	Description	Weight
Simple	Simple user interface. Touches only a single database entity. Its success scenario has three steps or less. Its implementation involves less than five classes.	5
Average	More interface design. Touches two or more database entities. Between four and seven steps. Its implementation involves between five and 10 classes.	10
Complex	Complex user interface or processing. Touches three or more database entities. More than seven steps. Its implementation involves more than 10 classes.	15

large number of steps can be reduced without affecting the business process.

The UUCW is calculated by tallying the number of use cases in each category, multiplying each total by its specified weighting factor, and then adding the products. For example, Table 2 computes the UUCW for the sample case study.

UAW

In a similar manner, the Actor Types are classified as simple, average, or complex as shown in Table 3.

The UAW is calculated by totaling the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the products. Table 4 computes the UAW for the sample case study.

The UUCP is computed by adding the UUCW and the UAW. For the data used in Tables 2 and 4, the UUCP = 210 + 12 = 222.

The UUCP is *unadjusted* because it does not account for the TCFs and ECFs.

TCFs

Thirteen standard technical factors exist to estimate the impact on productivity that various technical issues have on a project (see Table 5, page 20). Each factor is weighted according to its relative impact.

For each project, the technical factors are evaluated by the development team and assigned a *perceived complexity* value between zero and five. The perceived complexity factor is subjectively determined by the development team's perception of the project's complexity – concurrent applications, for example, require more skill and time than single-threaded applications. A perceived complexity of 0 means the technical factor is irrelevant for this project, 3 is average, and 5 is strong influence. When in doubt, use 3.

Each factor's weight is multiplied by its perceived complexity factor to produce the *calculated factor*. The calculated factors are summed to produce the *Technical Total Factor*. Table 6 (see page 20) calculates the technical complexity for the case study.

Two constants are computed with the Technical Total Factor to produce the TCF. The constants constrain the effect the TCF has on the UCP equation from a range of 0.60 (perceived complexities all zero) to a maximum of 1.30 (perceived complexities all five).

TCF values less than one reduce the UCP because any positive value multiplied by a positive fraction decreases in magnitude: 100 * 0.60 = 60 (a reduction of 40 percent).

TCF values greater than one increase the UCP because any positive value multiplied by a positive mixed number increases in magnitude: 100 * 1.30 = 130 (an increase of 30 percent).

Since the constants constrain the TCF from a range of 0.60 to 1.30, the TCF can impact the UCP equation from -40 percent (.60) to a maximum of +30 percent (1.30).

For the mathematically astute, the complete formula to compute the TCF is:

$$TCF = C_1 + C_2 \sum_{i=1}^{13} W_i * F_i$$

where,

Constant 1 (C_1) = 0.6
Constant 2 (C_2) = .01
W = Weight
F = Perceived Complexity Factor

For the rest of us, a more digestible equation is:

TCF = 0.6 + (.01 * Technical Total Factor)

For Table 6, the TCF = 0.6 + (0.01 * 19.5) = 0.795, *resulting in a reduction of the UCP by 20.5 percent.*

ECFs

The ECF (see Table 7, page 21) provides a concession for the development team's experience. More experienced teams will have a greater impact on the UCP computation than less experienced teams.

The development team determines each factor's *perceived impact* based on its perception the factor has on the project's

Table 2: *Computing UUCW*

Use Case Type	Description	Weight	Number of Use Cases	Result
Simple	Simple user interface. Touches only a single database entity. Its success scenario has three steps or less. Its implementation involves less than five classes.	5	7	35
Average	More interface design. Touches two or more database entities. Between four and seven steps. Its implementation involves between five and 10 classes.	10	13	130
Complex	Complex user interface or processing. Touches three or more database entities. More than seven steps. Its implementation involves more than 10 classes.	15	3	45
			Total UUCW	210

Table 3: *Actor Classifications*

Actor Type	Description	Weight
Simple	The actor represents another system with a defined application programming interface.	1
Average	The actor represents another system interacting through a protocol, like Transmission Control Protocol/Internet Protocol.	2
Complex	The actor is a person interacting via a graphical user interface.	3

Table 4: *Computing UAW*

Actor Type	Description	Weight	Number of Actors	Result
Simple	The actor represents another system with a defined application programming interface.	1	0	0
Average	The actor represents another system interacting through a protocol, like Transmission Control Protocol/Internet Protocol.	2	0	0
Complex	The actor is a person interacting via an interface.	3	4	12
			Total UAW	12

Technical Factor	Description	Weight
T1	Distributed System	2
T2	Performance	1
T3	End User Efficiency	1
T4	Complex Internal Processing	1
T5	Reusability	1
T6	Easy to Install	0.5
T7	Easy to Use	0.5
T8	Portability	2
T9	Easy to Change	1
T10	Concurrency	1
T11	Special Security Features	1
T12	Provides Direct Access for Third Parties	1
T13	Special User Training Facilities Are Required	1

Table 5: *Technical Complexity Factors*

success. A value of 1 means the factor has a strong, negative impact for the project; 3 is average; and 5 means it has a strong, positive impact. A value of zero has no impact on the project's success. For example, team members with little or no motivation for the project will have a strong negative impact (1) on the project's success while team members with strong object-oriented experience will have a strong, positive impact (5) on the project's success.

Each factor's weight is multiplied by its perceived impact to produce its *calculated factor*. The calculated factors are summed to produce the *Environmental Total Factor*.

Larger values for the Environment Total Factor will have a greater impact on the UCP equation.

Table 8 calculates the environmental factors for the case study project.

To produce the final ECF, two constants are computed with the Environmental Total Factor. The constants, "based on interviews with experienced Objectory users at Objective Systems" [1], constrain the impact the ECF has on the UCP equation from 0.425 (Part-Time Workers and Difficult Programming Language = 0, all other values = 5) to 1.4 (perceived impact all 0). Therefore, the ECF can reduce the UCP by 57.5 percent and increase the UCP by 40 percent.

The ECF has a greater potential impact on the UCP count than the TCF. The formal equation is:

$$ECF = C_1 + C_2 \sum_{i=1}^{8} W_i * F_1$$

where,

Constant 1 (C_1) = 1.4
Constant 2 (C_2) = -0.03
W = Weight
F = Perceived Impact

Informally, the equation works out to be:

ECF = 1.4 + (-0.03 * Environmental Total Factor)

Table 6: *Calculating the Technical Total Factor*

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight* Perceived Complexity)
T1	Distributed System	2	1	2
T2	Performance	1	3	3
T3	End User Efficiency	1	3	3
T4	Complex Internal Processing	1	3	3
T5	Reusability	1	0	0
T6	Easy to Install	0.5	0	0
T7	Easy to Use	0.5	5	2.5
T8	Portable	2	0	0
T9	Easy to Change	1	3	3
T10	Concurrency	1	0	0
T11	Special Security Features	1	0	0
T12	Provides Direct Access for Third Parties	1	3	3
T13	Special User Training Facilities Are Required	1	0	0
			Technical Total Factor	19.5

For the sample case study, the author's software development experience resulted in a high ETF. The most significant negative factor was the author's lack of experience in the application domain.

For Table 8, the ECF = 1.4 + (-0.03 *26) = 0.62, *resulting in a decrease of the UCP by 38 percent.*

Calculating the UCP

As a reminder, the UCP equation is:

UCP = UUCP * TCF * ECF

From the above calculations, the UCP variables have the following values:

UUCP = 222
TCF = 0.795
ECF = 0.62

For the sample case study, the final UCP is the following:

UCP = 222 * 0.795 * 0.62

UCP = 109.42 or 109 use case points

*Note for the sample case study, the TCF and ECF reduced the UUCP by approximately 49 percent (109/222*100).*

By itself, the UCP value is not very useful. For example, a project with a UCP of 222 may take longer than one with a UCP of 200, but we do not know by how much. Another factor is needed to estimate the number of hours to complete the project.

PF

The PF is the *ratio of development man-hours needed per use case point.* Statistics from past projects provide the data to estimate the initial PF. For instance, if a past project with a UCP of 120 took 2,200 hours to complete, divide 2,200 by 120 to obtain a PF of 18 man-hours per use case point.

Estimated Hours

The total estimated number of hours for the project is determined by multiplying the UCP by the PF.

Total Estimate = UCP * PF

If no historical data has been collected, consider two possibilities:

1. Establish a baseline by computing the UCP for previously completed projects (as was done with the sample case study in this article).

2. Use a value between 15 and 30 depending on the development team's overall experience and past accomplishments (Do they normally finish on time? Under budget? etc.). If it is a brand-new team, use a value of 20 for the first project.

After the project completes, divide the number of actual hours it took to complete the project by the number of UCPs. The product becomes the new PF.

Since the sample case study presented in this article actually took 990 hours to complete, the PF for the next project is: 990/109 = 9.08

Industry Case Studies

From the time Karner produced his initial report in 1993, many case studies have been accomplished that validate the reasonableness of the UCP method.

In the first case study in 2001, Suresh Nageswaran published the results of a UCP estimation effort for a product support Web site belonging to large North American software company [4]. Nageswaran, however, extended the UCP equation to include testing and project management coefficients to derive a more accurate estimate.

While testing and project management might be considered non-functional requirements, nevertheless they can significantly increase the length of the project. Testing a Java 2 Enterprise Edition implementation, for example, may take longer than testing a Component Object Model component; it is not unusual to spend significant time coordinating, tracking, and reporting project status.

Nageswaran's extensions to the UCP equation produced an estimate of 367 man-days, a deviation of 6 percent of the actual effort of 390 man-days.

In a recent e-mail exchange with this author, Nageswaran said he had also applied the UCP method to performance testing, unit-level testing, and white box testing.

In the second case study, research scientist Dr. Bente Anda [5] evaluated the UCP method in case studies from several companies and student projects from the Norwegian University of Science and Technology that varied across application domains, development tools, and team size. The results are shown in Table 9.

For the above studies, the average UCP estimate is 19 percent; the average expert estimate is 20 percent.

Additionally, at the 2005 International Conference on Software Engineering, Anda, et al. [6] presented a paper that described the UCP estimate of an incremental, large-scale development project that was within 17 percent of the actual effort.

In the third case study, Agilis Solutions and FPT Software partnered to produce an estimation method, based on the UCP method that produces very accurate estimates. In an article that was presented at the 2005 Object-Oriented, Programming, Systems, Languages, and Applications conference, Edward R. Carroll of Agilis Solutions stated:

> After applying the process across hundreds of sizable (60 man-months average) software projects, we have demonstrated metrics that prove an estimating accuracy of less than 9 percent deviation from actual to estimated cost on 95 percent of our projects. Our process and this success factor are documented over a period of five years, and across more than 200 projects. [7]

To achieve greater accuracy, the Agilis Solutions/FPT Software estimation method includes a risk coefficient with the UCP equation.

Environmental Factor	Description	Weight
E1	Familiarity With UML*	1.5
E2	Part-Time Workers	-1
E3	Analyst Capability	0.5
E4	Application Experience	0.5
E5	Object-Oriented Experience	1
E6	Motivation	1
E7	Difficult Programming Language	-1
E8	Stable Requirements	2

*Note: Karner's original factor, "Familiar with Objectory," was changed to reflect the popularity of UML.

Table 7: *Environmental Complexity Factors*

Conclusion

An early project estimate helps managers, developers, and testers plan for the resources a project requires. As the case studies indicate, the UCP method can produce an early estimate within 20 percent of the actual effort, and often, closer to the actual effort than experts and other estimation methodologies [7].

Moreover, in many traditional estima-

Table 8: *Calculating the Environmental Total Factor*

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor (Weight*Perceived Complexity)
E1	Familiarity With UML	1.5	5	7.5
E2	Part-Time Workers	-1	0	0
E3	Analyst Capability	0.5	5	2.5
E4	Application Experience	0.5	0	0
E5	Object-Oriented Experience	1	5	5
E6	Motivation	1	5	5
E7	Difficult Programming Language	-1	0	0
E8	Stable Requirements	2	3	6
	Environmental Total Factor			26

Table 9: *Use Case Studies*

Company	Project	Use Case Estimate	Expert Estimate	Actual Effort	Deviation Use Case Estimate	Deviation Expert Estimate
Mogel	A	2,550	2,730	3,670	-31%	-26%
Mogel	B	2,730	2,340	2,860	-5%	-18%
Mogel	C	2,080	2,100	2,740	-24%	-23%
CGE and Y	A	10,831	7,000	10,043	+8%	-30%
CGE and Y	B	14,965	12,600	12,000	+25%	+5%
IBM	A	4,086	2,772	3,360	+22%	-18%
Student Project	A	666		742	-10%	
Student Project	B	487		396	+23%	
Student Project	C	488		673	-25%	

tion methods, influential technical and environmental factors are not given enough consideration. The UCP method quantifies these subjective factors into equation variables that can be tweaked over time to produce more precise estimates.

Finally, the UCP method is versatile and extensible to a variety of development and testing projects. It is easy to learn and quick to apply.

The author encourages more projects to use the UCP method to help produce software on time and under budget.◆

References

1. Karner, Gustav. "Resource Estimation for Objectory Projects." Objective Systems SF AB, 1993.
2. Albrecht, A.J. Measuring Application Development Productivity. Proc. of IBM Applications Development Symposium, Monterey, CA, 14-17 Oct. 1979: 83.
3. Jacobson, I., G. Booch, and J. Rumbaugh. The Objectory Development Process. Addison-Wesley, 1998.
4. Nageswaran, Suresh. "Test Effort Estimation Using Use Case Points." June 2001 <www.cognizant.com/cogcommunity/presentations/Test_Effort_Estimation.pdf>.
5. Anda, Bente. "Improving Estimation Practices By Applying Use Case Models." June 2003 <www.cognizant.com/cogcommunity/presentations/Test_Effort_Estimation.pdf>.
6. Anda, Bente, et al. "Effort Estimation of Use Cases for Incremental Large-Scale Software Development." 27th International Conference on Software Engineering, St Louis, MO, 15-21 May 2005: 303-311.
7. Carroll, Edward R. "Estimating Software Based on Use Case Points." 2005 Object-Oriented, Programming, Systems, Languages, and Applications (OOPSLA) Conference, San Diego, CA, 2005.

About the Author

Roy K. Clemmons is an employee of Diversified Technical Services, Inc. He has more than 20 years experience in software design and development. Currently, he is contracted to the Retrieval Applications Group at Randolph Air Force Base, Texas, where he works on the Virtual Military Personnel Flight system and the Retrieval Applications Web site.

Diversified Technical Services, Inc.
403 E Ramsey STE 202
San Antonio, TX 78216
Phone: (210) 565-1119
E-mail: roy.clemmons.ctr@randolph.af.mil

MORE ONLINE FROM CROSSTALK

CROSSTALK is pleased to bring you this additional article with full text at <www.hill.af.mil/crosstalk/2006/02/index.html>.

Hard Skills Simulations: Tackling Defense Training Challenges Through Interactive 3-D Solutions

Josie Simpson
NGRAIN Corporation

The defense industry today faces a number of challenges around skills training, primarily driven by an increased pace of operations, the growing need to cross-train technical personnel to meet mission objectives, and ever-shrinking training budgets. Combined, these challenges can be daunting; but they can be overcome through the insertion of advanced technologies in instructional programs. Until recently, the use of three-dimensional (3-D) in hard skills training was limited to high-end applications such as flight simulators. Today, new technologies have been introduced that remove the traditional barriers to 3-D, allowing interactive 3-D to be used in lower-end applications, including maintenance training. Hard skills simulations, most notably 3-D virtual equipment, provide an innovative new way to cost-effectively train students to standard in less time on maintenance procedures and repair tasks, while simultaneously helping to improve performance in the field through on-the-job training aids. The result is reduced costs and a higher level of preparedness, ultimately saving lives.

Software Estimating Models: Three Viewpoints

Dr. Randall W. Jensen
Software Technology Support Center

Lawrence H. Putnam Sr.
Quantitative Software Management, Inc.

William Roetzheim
Cost Xpert Group, Inc.

This article compares the approaches taken by three widely used models for software cost and schedule estimation. Each of the models is compared to a common framework of first-, second-, and third-order models to maintain consistency in the comparisons. The comparisons illustrate significant differences between the models, and show significant differences in the approaches used by each of the model classes.

The purpose of this article is to present an unbiased comparison of three approaches to estimating software development costs. Rather than a single author attempting to arrive at a middle of the road, politically correct description of the three approaches, this article presents the comparison according to three individuals who are at the heart of these major estimating philosophies (from the horses' mouths so to speak).

Origins and Evolution of Software Models

This article prompted an enlightening trip back into the fuzzy history of computer-based software cost and schedule estimating methods and tools. It appears that the origins are not that remote, and the methods appeared over a relatively short period of time and have evolved in spurts and starts since then. The first real contribution to the estimating technology happened in 1958 with the introduction of the Norden staffing profile [1]. This profile has been incorporated in many of the estimating methodologies introduced since then.

A flurry of software estimating methods was introduced beginning in the mid-1970s and throughout the next decade. The first publication of any significance was presented by Ray Wolverton [2] of TRW in 1974. Wolverton was a major contributor to the development of the Constructive Cost Model (COCOMO) [3]. The second major contribution to the evolution of software estimating tools was the Doty Associates model [4], developed for the U.S. Air Force in 1977. The period from 1974 through 1981 brought most of the software estimating models (tools) we use today to the marketplace.

Each of these tools evolved at a gentle pace (refined algorithms and drivers) until about 1995, when significant changes were made to many of the models. COCOMO II, for example, had several releases between 1995 and 1999. Sage, released in 1995, is a major redefinition of the 1979 Seer model. Cost Xpert, introduced in 1996, is a major modification of the COCOMO family line. It is amazing that, in the 25 years elapsed since the first wave of estimating tools, development environments have changed so little that these models and their predicted environments are still valid today.

Framework for Discussion

When we look at software estimating models, they generally fall into one of three classes: first-, second- or third-order forms. This article compares three widely used models using the three classes as a framework for discussion and comparison.

First-Order Model

The first-order model is the most rudimentary model class. The model is simply a productivity constant, defining the production capability of the development organization in terms of arbitrary production units multiplied by the software product effective size to obtain the development effort or cost. The production units can be source lines of code (SLOC), function points (FPs), object points, use cases, and a host of other units. For the purpose of this discussion, we will use effective source lines of code (ESLOC) as the production measure, and person-hours per ESLOC as the productivity measure. This can be stated as follows:

$$E_d = C_k S_e \quad (1)$$

where,

E_d is the development effort in person hours (ph).
C_k is a productivity factor (ph/esloc).
S_e is the number of ESLOC.

The productivity factor is commonly determined by the product type, historic developer capability, or both as derived from past projects. As simple as this equation is, it is widely used to produce high-level, rough estimates. An expansion of this form used as far back as the 1970s is:

$$E_d = C_k(S_{new} + 0.75S_{modified} + 0.2S_{reused})\ ph \quad (2)$$

Or it is a similar variation. The weakness of this model is its insensitivity to the magnitude of the effective product size. Productivity is, or at least should be, decreased for larger projects.

Second-Order Model

The second-order model compensates for the productivity decrease in larger projects by incorporating an *entropy* factor to account for the productivity change. The entropy effect demonstrates the impact of a large number of communications paths that are present in large development teams. The number of paths is specified by n(n-1)/2 where n is the number of development personnel. The second-order model becomes the following:

$$E_d = C_k S_e^{\beta} \quad (3)$$

where,

ß is an entropy factor that accounts for the productivity change as a function of effective product size.

An entropy factor value of 1.0 represents no productivity change with size. An entropy value of less than 1.0 shows a productivity increase with size, and a value greater than 1.0 represents a productivity decrease with size. Entropy values of less than 1.0 are inconsistent with historical software data[1]. Most of the widely used models in the 1980s (COCOMO embedded mode, PRICE-S, REVIC, Seer, and SLIM) used entropy values of approximately 1.2 for Department of Defense projects.

The major weakness of this model is

its inability to adjust the productivity factor to account for variations between projects in development environments. For example, contractor A may have a more efficient process than contractor B; however, contractor A may be using a development team with less experience than used in the historic productivity factor. Different constraints may be present in the current development than was present in previous projects. In addition, using a fixed, or calibrated, productivity factor limits the model's application across a wide variety of environments.

Third-Order Model

The third-order model compensates for the second-order model's narrow applicability by incorporating a set of environment factors to adjust the productivity factor to fit a larger range of problems. The form of this model is as follows:

$$E_d = C_k \left(\prod_{i=1}^{n} f_i\right) S_e^{\beta} \qquad (4)$$

where,

f_i is the ith environment factor.
n is the number of environment factors.

The number of environment factors varies across estimating models, and is typically between 15 and 32.

Using the generalized model set defined in equations (1) through (4), we have a framework for comparing the definition and features of the software estimating models. Three model types are described and compared in the following sections of this article: (1) models evolving from the 1979 Seer model developed and described by Dr. Randall Jensen, (2) models evolving from the COCOMO model described by William Roetzheim, and (3) the SLIM model developed and described by Lawrence Putnam, Sr.

Sage/Seer Effort and Schedule Calculations

Software development involves three important elements: people, a process, and a product. The people element describes management approach and style as well as personnel attributes, including capability, motivation, and communication effectiveness. Process represents the software development approach, tools, practices, and life-cycle definition. The product attributes include project-imposed constraints such as development standard, memory and time constraints, and security issues.

Software development is the most communication-intensive of all engineering processes. This unique software process characteristic suggests significant productivity gains are more likely to be realized through communication improvement, rather than through technology. Communication effectiveness is determined by organizational structure, management approach, and development environment. The Jensen model [5], and its implementations, embodies the impacts of the three important elements in software development cost and schedule estimates.

This section of the article discusses the underlying theory of a line of software cost and schedule estimating tools that evolved from the Jensen software model [6] at Hughes Aircraft Company's Space and Communications Group in 1979. The original model implementation was called Seer (not an acronym), a name later converted to the acronym SEER-SEM (Software Evaluation and Estimation of Resources-Software Estimating Model) [7] and trademarked by Galorath Associates, Inc. (GAI) in 1990. The Seer concepts were influenced by Lawrence Putnam's work [8] published in 1976 and the Doty Associates [9] estimating model published in 1977. The Seer model was derived from the U.S. Army data used by Putnam to develop SLIM, but with a different interpretation of the data itself.

The first major update to the Jensen model came in 1995 (Jensen II) with the addition of project management and personnel characteristics effects to the model. The impacts of motivation, management style, and teaming on productivity have long been suspected, but until 1995 the data to support the alleged behavior was simply insufficient to make credible model changes. These changes were implemented in the Software Engineering, Inc., Sage [10] software estimating system. For the sake of brevity, the Jensen model (I and II) will be referred to as Sage throughout this discussion. The following discussion applies to all descendants of the original Jensen estimating model. The fundamental equations are the following:

$$S_e = C_{te} \sqrt{K} T_d \qquad (5)$$

and

$$D = \frac{K}{T_d^3} \qquad (6)$$

where,

C_{te} is the effective technology constant of the development activity.
K is the total life-cycle effort in person-years (py) of the software development starting with the software requirements review through the software's end of life.
T_d is the software product development time in years.
D is the product complexity rating.

Equations (5) and (6) solved simultaneously provide both schedule and effort estimates in one calculation with the relationship between effort and schedule linked by the product complexity. This approach is unique to the Jensen and Putnam models.

The parameter D is the same as the *Difficulty* parameter discovered by Putnam. This explains the use of D to describe what the Jensen model refers to as complexity.

Development effort is defined by

$$E_d = 0.3945K$$

where,

E_d is the development effort in py through the final qualification test.

The Sage software development effort equation is an implementation of the third-order model discussed in the introduction to the model comparison even though it is not immediately apparent. Combining equations (5) and (6), we find the following:

$$E_d = \frac{18{,}797 D^{0.4}}{C_{te}^{1.2}} S_e^{1.2} \text{ person months (pm)} \qquad (7)$$

The ugly part of equation (7) preceding the effective size element comprises the productivity factor of the third order model. The effective technology constant C_{te} contains two components: (1) the basic technology constant C_{tb} representing the development organization's raw capability; that is, capability independent of the constraints imposed by a specific develop-

Table 1: *Sage Analyst Capability Ratings*

Definition	Highly motivated AND experienced team organization	Highly motivated OR experienced team organization	Traditional software development organization	Poorly motivated OR non-associative organization	Poorly motivated AND non-associative organization
Relative cost impact	0.71	0.86	1.00	1.19	1.46

ment, and (2) the impacts of 24 environment factors/constraints. The C_{te} value is obtained from the following:

$$C_{te} = \frac{C_{tb}}{\prod_{i=1}^{24} f_i} \quad (8)$$

where,

***C_{tb}* represents the basic technology constant.**
***f_i* is the ith product-impacted environment factor.**

The C_{tb} value can be anywhere between 2,000 and 20,000 with a normal range between 5,500 and 7,500. The highest value observed from available data at this time is about 8,635. Higher values obviously imply higher productivity and efficiency. Theoretical values of C_{te} range from 0 through 20,000. The practical upper C_{te} limit is defined by an organization's rating. The practical lower C_{te} bound is about 500 for a less-than-average organization and severe product constraints.

The relative cost impact of the analyst capability rating for Sage is shown in Table 1 as an example of one of the 24 environment factors.

The product development time T_d is the minimum development time as can be seen from the Paul Masson cost-time relationship shown in Figure 1. The *Paul Masson Point* represents the minimum development time that can be achieved with a specified size, environment, and complexity. By attempting to reduce the schedule below the minimum time, the cost will increase and the schedule will also increase as described by Brooks Law [11]: "Adding people to a late software project makes it later." Sage computes the minimum development schedule as a default.

The region shown by the double-headed arrow is represented as a square-law relationship in the model between cost and schedule, or $c = KT_d^2$. At first glance it seems that a longer schedule should equate to higher cost. By explaining the phenomenon in two logical steps, the productivity gain over the region becomes clear. A longer schedule requires a smaller development team. A smaller team is more efficient; thus, productivity improves and the cost decreases. This phenomenon applies until the productivity gain is eaten up by fixed costs.

As an example of the effort and schedule predicted by Sage, let us assume the following development project parameters: The product is a satellite mission operations system (application with significant logical complexity with some changes to the underlying operating system) consisting of 59,400 new SLOC. A basic technology rating of 7,606 places the developer at the upper end of the typical capability range. The project environment constraints, including the Institute of Electrical and Electronics Engineers Standard 12207 development standard, reduce the effective technology rating to 2,624. The third-order equation form of equation (7) reduces to $E_d = 4.007S_e^{1.2}$. The results are tabulated in Table 2.

The minimum development schedule, which is simultaneously calculated in equation (5), is approximately 25 months.

Quantitative Software Management View of Software Estimating and Productivity Measurement

Productivity measurement is used to:

1. Tune estimating systems.
2. Baseline and measure progress in software development.

But what is *productivity*? My answer is: It is not SLOC/PM or FP/PM. This is the traditional view from economic theory – output/input.

The software industry has 35 years of experience that shows that this ratio works poorly as a productivity metric. At Quantitative Software Management (QSM), we have learned why it does not work: because it ignores schedule, and software development is very sensitive to schedule. A vast amount of our 27-year collection of data, some 6,600 completed systems, coupled with empirical analysis shows that schedule is the most important factor in estimating relationships and must be dealt with explicitly. If schedule is not so recognized and dealt with, then it will assert itself implicitly and cause much grief. By this, I mean both time and effort must be included multiplicatively in an expression for a good software algorithm. We have found this to be of the conceptual form:

Amount of function = Effort * Schedule * Process Productivity (9)

where,

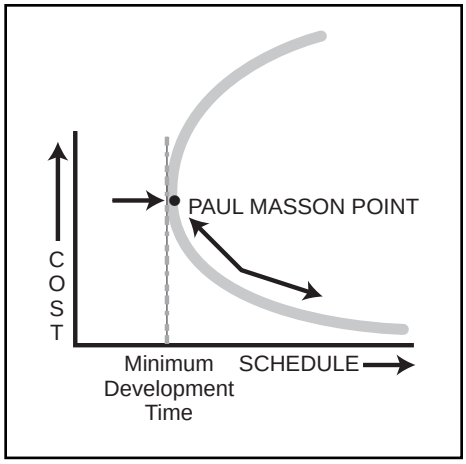


Figure 1: *Paul Masson Rule*

Effort and Schedule have exponents. Specifically,

Size = (Effort/ßeta)$^{1/3}$ Schedule$^{4/3}$ Process Productivity Parameter (10)

where,

***Size* is the size in SLOC, or other measure of amount of function.**
***Effort* is the amount of development effort in py.**
***ßeta* is a special skills factor that varies as a function of size from 0.16 to 0.39.**
***ßeta* has the effect of reducing process productivity, for estimating purposes, as the need for integration, testing, quality assurance, documentation, and management skills grows with increased complexity resulting from the increase in size.**
***Schedule* is the development time in years.**
***Process Productivity Parameter* is the productivity number that we use to tune the model to the capability of the organization and the difficulty of the application. We do this by calibration as explained in the next section. The theoretical range of values is from 610 to 1,346,269. The range of values seen in practice across all application types is 1,974 to 121,393 and varies exponentially.**

Estimating and Tuning Models

All models need tuning to moderate the *productivity* adjusting factor.

- Some models use effort multipliers or modifiers.
- I have found that we can use the software equation to *calibrate* our estimat-

Table 2: *Example Estimate*

S_e kesloc	*D*	C_{tb}	C_{te}	E_d . pm	T_d mo	*Productivity*, sloc/pm
59.4	12	7,606	2,624	538.7	25	110

ing algorithm. This calibration process is far more accurate than intelligent guesses of the settings for effort multipliers because it is based on real data from the development organization. All we need to do is to rearrange the software equation into this form:

$$\text{Process Productivity Parameter} = \text{Size} / ((\text{Effort}/\beta\text{eta})^{1/3} (\text{Schedule}^{4/3})) \quad (11)$$

From historic projects we know the size (SLOC, FPs, etc.), effort (py) and schedule (years). Then just put in a consistent set of historic numbers and we can calculate a Process Productivity Parameter. This works well. Note that the expression for Process Productivity Parameter includes schedule and that it is multiplicatively tied to effort. This expression is our definition of software productivity.

This software equation has two variables that we want to solve for: schedule and effort. That means we have to have another equation to get a solution in the form of a schedule-effort pair. The second equation may be in the form of a constraint like maximum budget for the project (Maximum Development Effort = Maximum Cost/$Average Burdened Labor Rate), or, Maximum Schedule = Maximum Development Time in years. There are a number of other constraints that can be used such as peak manpower, or maximum manpower buildup rate (defined as *Difficulty* in Randall Jensen's preceding section).

Example of an Estimate

Here is a simple example: We need an estimate for a Global Positioning System navigation system for an air-launched, land-attack missile. We have completed the high-level design phase. Estimated size is 40,000 C++ SLOC; Process Productivity Parameter for this class of work (real time avionic system) is 3,194 [taken from Table 14.8, in 12], *ßeta* = 0.34 [taken from Table 14.4, in 13]. We have to deliver the system to the customer for operational service in two years (24 months from the end of high-level design) The fully burdened contractor labor rate is $200,000 per person-year. Substituting in the software equation and solving for effort, we have the following:

$$40{,}000 = (\text{Effort}/0.34)^{1/3}\, 2^{4/3}\, 3{,}194$$
$$\text{Effort} = 41.74 \text{ PY (Approximately 500.9 pm)}$$
$$\text{Cost} = \$200{,}000/\text{PY} * 41.74 \text{ PY} = \$8.35 \text{ million}$$

Observations Concerning Effort Multipliers/Moderators

Many estimating systems use a family of adjusting factors to try to tune their productivity constant. Between 15 and 25 of these *tweekers* are typical. The values are picked from a scale centered on 1.0 that increase or decrease the Productivity constant. This process does moderate the baseline productivity value. Unfortunately, it is highly subjective – dependent upon the judgment of the practitioner. It is not consistent or reproducible from person to person and hence it introduces considerable uncertainly into the estimate that follows.

At QSM, we use a family of tweekers for tools and methods, technical complexity of the project, competence, experience, and skill of the development team. This list is similar to most other estimating systems. But we use it only as a *secondary* technique for those organizations that truly have no historic data. The notion of calibration from historic data is much more accurate and powerful because it captures the real capability and character of the development organization in a single number – the Process Productivity Parameter in the software equation. This single number is unambiguous and consistent; there is no subjectivity involved.

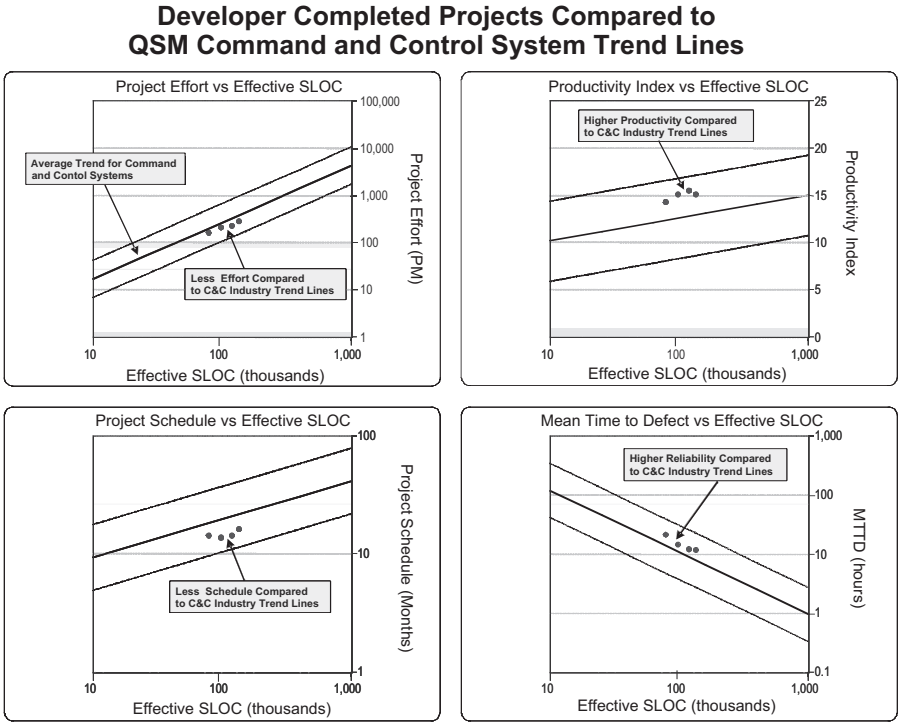
Benchmarking

One of the nice things about being able to substitute your historical data into the software equation is that you can calculate an unambiguous number that can be used for benchmarking. We transform the Process Productivity into a linear scale and call that a Productivity Index (PI). If we collect a homogeneous set of data from a development organization and calculate the PI for each project, we see a fairly normal distribution with a central tendency.

The central tendency represents our current average PI. If we keep track of all our projects over time and if we are doing process improvement (trying to move up the SEI scale) then we will see an increase in the PI over time. Often we can plot the PI behavior over time and pick up the trend. Moreover, this graphical approach makes it easy to compare organizations doing similar types of work. Extending this thinking a little bit provides the ability to quantitatively compare the real capability of bidders on a software development contract. This comparison process takes a lot of guesswork out of trying to determine the real capability of vendors.

The benchmarking idea can be extended easily to show performance capabilities of the development organization. The idea is to take a fairly large body of contemporaneous historic data from the same industry sector, then generate some trend lines plots of the generic form: management metric (schedule, effort, staffing, defects, etc.) versus size (SLOC, FPs, objects, etc.). Next, superimpose data points from the development

Figure 2: *Trend Lines Plots*



organization being measured on top of these trend lines and see how they position (high or low) compared with the industry average trend line at the appropriate size. For example, if your data shows a pattern of falling below the average line for effort, schedule, and defects you are a more effective producer (high productivity developer). Almost invariably your PI and Mean Time to Defect (MTTD) will be higher as well. This means you can unambiguously and quantitatively measure productivity. After nearly 30 years of experience doing it, we know it works consistently and well. An example of such plots is shown in Figure 2.

Cost Xpert Effort and Schedule Calculations

Algorithmically, Cost Xpert started with the COCOMO models, added Revised Intermediate COCOMO (REVIC) extensions, and then layered functionality on top of this base. The core approach of Cost Xpert is as follows:

1. Define scope using various measures of size (e.g., SLOC, FPs, class-method points, user stories, many others) for new development, reused code, and commercial off-the-shelf (COTS) components).
2. Use scope and a variety of adjusting factors (addressed below) to calculate effort.
3. Use effort to calculate optimal schedule.
4. Feedback deviations from the optimal schedule to adjust effort, if necessary.

First-Order Modeling in Cost Xpert

First order modeling involves linear calculations of effort using a productivity constant:

$$\text{Effort} = \text{Productivity} * \text{Size} \qquad (12)$$

In Cost Xpert, all sizing metrics (SLOC, FPs, class-method points, user stories, etc.) and development stage (new, reused, COTS) are normalized to a SLOC equivalent. This SLOC equivalent is defined such that it is valid for estimating effort, although in some situations it may not accurately represent physical lines of code (for example, in environments where much of the code is auto-generated). In other words, although it once represented physical lines of code calculated using an approach called backfiring, and although that relationship is still true for older development environments, with newer environments it has become more of an estimating size proxy.

COCOMO and REVIC use(d) relatively small databases of projects and have a correspondingly small universe of productivity numbers. For example, COCOMO II uses the value 2.94 person-months per thousand SLOC (KSLOC) [14]. Commercial vendors, including the Cost Xpert Group, are able to maintain much larger databases of projects and hence can segment those databases into a finer granularity of project classes, each with a corresponding productivity number.

Cost Xpert uses a project type variable to denote the nature of the project (e.g., military, commercial, and internet) and set the coefficient for given historic database segments. Actual numbers for these sample project classes are shown in Table 3 [15]. The productivity number multiplied by KSLOC yields the first order effort in person-months.

However, Cost Xpert Group research has determined that there are additional sources of variation across projects and project domains (see the Third-Order Modeling section for examples). Our calibrations account for the additional sources of variation and therefore produce different coefficients for given project classes than the models with reduced factor sets. The overall net productivity in Cost Xpert thus accounts for second-order and third-order modeling described in the following sections.

Second-Order Modeling in Cost Xpert

Second-order modeling in Cost Xpert involves adjusting the productivity to allow for economies or diseconomies of scale. An economy of scale indicates that the productivity goes up as the scope goes up. For example, you would expect that it is cheaper per yard to install 100,000 yards of carpeting than 1,000 yards. A diseconomy of scale indicates that the productivity goes down as the scope goes up. In software, we are dealing with diseconomies (larger projects are less efficient). This is modeled by raising the size to a power, with a power greater than 1.0 increasing the apparent scope, and thereby effort, with increasing project size. The second-order model looks like this:

$$\text{Effort} = \text{Productivity Factor} * \text{Size}^{\text{Scaling Factor}} \qquad (13)$$

Table 4 shows some scaling factors by project type.

Third-Order Modeling in Cost Xpert

Third-order modeling in Cost Xpert involves adjusting the productivity and scaling factors by project-specific variables. Cost Xpert uses 32 variables, called environment variables (E) to adjust the productivity number up or down, and five variables, called scaling variables (S) to adjust the scaling factor up or down. Each variable is set to a value ranging from very low to extremely high using defined criteria for each setting. Many of these variables will typically be fixed at a given value for an organization, with only a handful actually varying from project to project. The total productivity factor adjustment (PFA) is the product of the individual productivity factor values.

Project Type	Productivity Factor (pm/KSLOC)
Military	3.97
Commercial	2.40
Internet	2.51

Table 3: *Productivity Factors*

Project Type	Scaling Factor
Military, Complex	1.197
Military, Average	1.120
Military, Simple	1.054
Commercial	1.054

Table 4: *Scaling Factors for Various Project Types*

$$\text{PFA} = \prod \text{E} \qquad (14)$$

Table 5 (see page 28) shows some sample environmental variables and the resultant PFAs as a sample of how this works.

The higher the number, the more effort required to deliver the same KSLOC, so in the Table 5 sample the commercial product would require less effort per KSLOC than the military project.

Additionally, Cost Xpert has introduced different types of linear factors for important sources of effort variation. We find that much of the variance between project classes (e.g., military versus commercial) can be accounted for by the life cycles and standards typically employed. We have introduced the following factors:

- **Project Life Cycle:** The life cycle (template of activities, or work breakdown structure) used for development).
- **Project Standard:** The deliverables (engineering specifications, or artifacts) produced during development.

These factors are rated by selecting named life cycles and standards, which are different than rating the standard environmental variables. In addition to adjusting

	Required Reliability (E)		Multi-Site Development (E)		Security Classification (E)		Net Productivity (PFA)
Military	Very High	1.26	Nominal	1.00	High	1.10	1.386
Commercial	Nominal	1.00	Very High	0.86	Nominal	1.00	0.86

Table 5: *Sample Productivity Factor Adjustments*

	Life Cycle and Multiplier		Standard and Multiplier		Net Productivity
Sample 1	Waterfall	1.01	Military-498	1.34	1.35
Sample 2	RAD	0.91	RAD	0.51	0.46

Table 6: *Life Cycle and Standard Adjustment Factors*

effort, the life cycle and standard are used to create specific detailed project plans and page size estimates. They are also used in our defect model. Examples using actual numbers for sample projects are shown in Table 6, where RAD is rapid application design. These two samples could represent military and commercial projects respectively.

The relative productivity difference between the samples due to these two factors would be 1.35/.46 = 2.9 or 290%.

The five scaling variables (S) (not shown) work in a somewhat analogous manner, but the five factors are summed to adjust the exponential factor that applies to the diseconomy of scale. The total third-order formula is then the following:

$$\text{Effort} = \prod \text{E} * \text{P} * \text{KSLOC}^{\text{Scaling Factor} + (\sum s/5)} \quad (15)$$

where,

Effort **is the effort in pm.**
E **are the various environmental factors.**
P **is the productivity factor (which is further broken down into a project type, life cycle, and standard).**
KSLOC **is the SLOC equivalent in thousands.**
S **are the five scaling factor adjustments.**
ScaleFactor **is the default scaling factor for this project type.**

If we use the military productivity factor from Table 3, military-complex from Table 4, the military PF adjustment from Table 5, and the life cycle and standard adjustments for sample 1 in Table 6, the equation simplifies to:

$$\text{Effort} = 3.97 * 1.386 * 1.35 * \text{KSLOC}^{(1.197 + (\sum s/5))} \quad (16)$$
$$\text{Effort} = 7.43 * \text{KSLOC}^{(1.197 + (\sum s/5))}$$

Calculating Schedule and Adjusting for Schedule Deviations

In Cost Xpert, the optimal schedule is driven by the calculated effort. The schedule formula is of the form:

$$\text{Schedule} = \propto * \text{Effort}^{\beta} \quad (17)$$

where,

Schedule **is the schedule in calendar months.**
∝ is a linear constant.
ß **is an exponential constant.**

Table 7 shows a couple of sample values.

Accelerating a project from this calculated schedule results in inefficiencies that then lower productivity and increase effort. Cost Xpert handles this schedule acceleration adjustment to productivity through a table lookup and interpolation between points in the table.

Table 7: *Sample Schedule Factors*

Project Type	α	ß
Military	3.80	0.378
Commerical	2.50	0.3348

Summary and Conclusions

The three model implementations described in this article represent the majority of the estimating approaches available to the estimator today. Our intent in this article is not to advocate a single software estimating approach or tool, but is to expose you, the reader, to the mindsets incorporated in the more widely used approaches available today.◆

References

1. Norden, P.V. "Curve Fitting for a Model of Applied Research and Development Scheduling." IBM Journal of Research and Development 2.3 (July 1958).
2. Wolverton, R.W. "The Cost of Developing Large-Scale Software." IEEE Transactions on Computers June 1974: 615-636.
3. Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
4. Herd, J.R., J.N. Postak, We. E. Russell, and K.R. Stewart. "Software Cost Estimation Study – Final Technical Report." Vol. I. RADC-TR-77-220. Rockville, MD: Doty Associates, Inc., June 1977.
5. Jensen, R.W. "Management Impact of Software Cost and Schedule." CROSSTALK July, 1996:6.
6. Jensen, R.W. A Macrolevel Software Development Cost Estimation Methodology. Proc. of the Fourteenth Asilomar Conference on Circuits, Systems and Computers. Pacific Grove, CA, 17-19 Nov. 1980.
7. Galorath, Inc. SEER-SEM Users Manual. El Segundo, CA: Galorath Inc., Mar. 2001.
8. Putnam, L.H. A Macro-Estimating Methodology for Software Development. Proc. of IEEE COMPCON '76 Fall, Sept. 1976: 138-143.
9. Herd, J.R., J.N. Postak, We. E. Russell, and K.R. Stewart. "Software Cost Estimation Study – Final Technical Report." Vol. I. RADC-TR-77-220. Rockville, MD: Doty Associates, Inc., June 1977.
10. Software Engineering, Inc. Sage User's Manual. Brigham City, UT: Software Engineering, Inc., 1995.
11. Brooks Jr., F.P. The Mythical Man-Month. Reading, MA: Addison-Wesley, 1975.
12. Putnam, Lawrence H., and Ware Myers. Measures for Excellence. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992: 237
13. Putnam. Measures for Excellence, 234.
14. Boehm B., et al. Software Cost Estimation With COCOMO II. Prentice-Hall, 2000.
15. Cost Xpert Group, Inc. Cost Xpert 3.3 Users Manual. San Diego, CA: Cost Xpert Group, Inc., 2003.

Note

1. This is a good place to point out a major difference between software and almost any other manufactured product. In other estimating areas, a large number of products improves productivity through the ability to spread costs over a large sample and reduce learning curve effects. The software product is but a single production item that becomes more complex to manage and develop as the effective size increases.

About the Authors

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, with more than 40 years of practical experience as a computer professional in hardware and software development. He developed the model that underlies the Sage and the Galorath, Inc. SEER-SEM software cost and schedule estimating systems. He retired as chief scientist in the Software Engineering Division of Hughes Aircraft Company's Ground Systems Group. Jensen founded Software Engineering, Inc., a software management-consulting firm in 1980. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He has a Bachelor of Science, a Master of Science, and a doctorate all in electrical engineering from Utah State University.

Software Technology Support Center
6022 Fir AVE BLDG 1238
Hill AFB, UT 84056
Phone: (801) 775-5742
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil

Lawrence H. Putnam Sr. is the founder and chief executive officer of Quantitative Software Management, Inc., a developer of commercial software estimating, benchmarking, and control tools known under the trademark SLIM. He served 26 years on active duty in the U.S. Army and retired as a colonel. Putnam has been deeply involved in the quantitative aspects of software management for the past 30 years. He is a member of Sigma Xi, Association for Computing Machinery, Institute of Electrical and Electronics Engineers (IEEE), and IEEE Computer Society. He was presented the Freiman Award for outstanding work in parametric modeling by the International Society of Parametric Analysts. He is the co-author of five books on software estimating, control, and benchmarking. Putnam has a Bachelor of Science from the United States Military Academy and a Master of Science in physics from the Naval Postgraduate School.

Quantitative Software Management, Inc.
2000 Corporate Ridge STE 900
McLean, VA 22102
Phone: (703) 790-0055
Fax: (703) 749-3795
E-mail: larry_putnam_sr@qsm.com

William Roetzheim is the founder of the Cost Xpert Group, Inc., a Jamul-based organization specializing in software cost estimation tools, training, processes, and consulting. He has 25 years experience in the software industry, is the author of 15 software related books, and over 100 technical articles.

2990 Jamacha RD STE 250
Rancho San Diego, CA 92019
Phone: (619) 917-4917
Fax: (619) 374-7311
E-mail: william@costXpert.com

Letter to the Editor

Dear CrossTalk Editor,

In the December 2005 issue, the article "Agile Software Development for the Entire Project" by Granville Miller, Microsoft, describes how the agile process in MSF can make the *agile* described in the Agile Manifesto <www.agilemanifesto.org> much easier to implement without all of those difficult changes that many others have experienced. He describes how these reflect the fine engineering practices at Microsoft that have led the MSF version of agile to already be a year late.

It has taken more than 20 years for parts of the American manufacturing industry to adopt lean thinking. Agile, which has many parallels to lean manufacturing, will also take a lot of effort and time. Change is always an effort, and only the dramatic benefits of agile make it worthwhile. Efforts by people like Granville Miller to water down agile by redefining the intent do not help. Efforts that add more process miss the point; process is defined by self-managing teams within frameworks. Decisions are made by these teams working closely with customers to maximize benefit and optimize results.

At the start of the agile movement, we were warned that the larger commercial interests would attempt to water it down to *fit* their existing tools. We should expect to see other similar fits such as from IBM (through RUP in the Eclipse Foundation) and others. The *refinements* suggested by Granville Miller do a disservice to everyone working on agile.

Ken Schwaber
Signatory to the Agile Manifesto
Founder of the Agile Alliance
Co-Author of the Scrum Agile process
ken.schwaber@controlchaos.com

Tech-Neologism

Language is a product of enlightened thinking. As such, one would think the evolution of language would be an open and exhilarating endeavor. However, in the early era of language, neologism was surprisingly lethargic and proscribed. For a fascinating insight into early linguistic expansion, I recommend Simon Winchester's book, "The Professor and the Madman: A Tale of Murder, Insanity and the Making of the Oxford English Dictionary."

In this medieval era, words had to pay their dues and patiently wait their turn before joining the literary league. Premodern academic prigs banded together to ward off the invasion of the unwashed word. Conservatism was favored over litheness.

In our post-modern world, the pendulum has swung in the opposite direction. Sports, musical, political, and urban lingo constantly alter, shape, and morph proper language. Who opened the gate for this new era of malleable idiom? Credit the engineers, scientists, and technologists born out of the scientific explosion impelled by two world wars.

Technical jargon turned to slang and slang made its way from military, scientific, and technical circles into the general vernacular. This process has gradually accelerated, reaching terminal velocity with the advent of the computer and internet. Our rapture, frustration, and superstition with technology invigorates a desire to name these new experiences, spawning new language. These new words and phrases are a great source of humor and intrigue. Here are a few techno-neologisms that tickle my fancy.

Let's start with an old favorite, polymorphism: a programming language's ability to process objects differently depending on their type or class. From its early days as the object-oriented world's buzzword of choice, polymorphism has all the makings of a classic word. Any word that starts with poly makes you smile. Morph brings a sense of intrigue – imagine the ability to appear in many forms. Most of our lives are spent changing form – polymorphism is the basis for the entire cosmetic industry. Ending with ism gives polymorphism a sense of permanence and historical significance. Since this is an old favorite, classify this word as a post-neologism.

How about honeymonkey? This sweet, yet playful, word emerged from Microsoft's need to sniff out malicious code. Its predecessor, the honeypot, sits on the Web as a server, attracting client-based malicious code. Honeymonkies take a reverse approach by surfing the net as a client, attracting server-based malicious code. Makes sense to me. You cruise the net offering honey and once you get a bite you start screeching like a monkey. Do you think Bill Gate's wife calls him honeymonkey?

Then there is Ohnosecond, the fraction of time it takes you to realize you have just goofed up, but cannot reverse your action. Around since the beginning of time, ohnosecond credits the premature use of the send button for its ascension to fame. Typically symbolized by putting your foot in your mouth, I am relieved to have a word that covers the situation: keeping my rima oris free of toe jam.

What is a Bi-stable Multivibrator? A flip-flop – the technical kind, not the flip-flops we wear on our feet. The shoe industry had to steal the term flip-flop from the tech industry because the underwear industry stole the term thong. To appease the CROSSTALK editors, I will refrain from further linking of words in this paragraph in any other way.

Gnutella. I love the word Gnutella. Gnutella allows users to share files in a truly distributed manner. However, it conjures up so much more: Marlon Brando yelling Gnutella, Jane Fonda floating in space, Godzilla's next foe, and a punk rock band. It will not be long before the number one name for babies will be Gnutella.

A sophomore favorite is Dag-tag. Dag-tag is an overly elaborate signature, quote, or ASCII art following an e-mail message. You will have to ask your Australian friends what a dag is. You can view dags on sheep as they walk away from you. Out west they are known as dingleberries. In general, undesirable stuff that is stuck in an inconvenient place is a dag-tag. That piece of toilet paper dangling from the boss's backside? Yes, that's a dag-tag.

An easy way to complete this article would be to cover all prefixed words. Words easily named by tacking a technical prefix to them, e.g., e-, cyber-, net-, info- and techno-. To me, that is cheating. However, there are three technical fields that are rapidly merging to form an interesting new field. That is the convergence of biology, information technology, and nanotechnology: Bio-Info-Nano. I propose a new prefix for this technology convergence: Bino-, which rhymes with the city Reno. I predict we will be drunk with its pervasiveness. We will create a game called Bino-, dogs will be named after Bino-, and we will sing about it in pre-school.

Googolplex was a benign scientific term representing the number 10 raised to the power of one googol, or (10^(10^100)), which is the number 1 followed by 10^100 zeros. With the massive popularity of a certain search engine, googolplex now conjures up a new vision: I see a massive cyber strip mall where you can Google and be Googled to your heart's desire.

Have you seen the acronym DSTN on your LCD screen? Do you know what it stands for? Double-layer Super Twist Nematic. This was obviously the inspiration for the cants used to order coffee at Starbucks. Dude, I'll have a Double-Layer Super Twist Nematic Latte, pronto.

My favorite technical word is Electrowetting. I am not kidding. This is a legitimate word and is a promising technology for optical switching networks and focusing lasers. Electrowetting (stop giggling) uses electrical fields to modify dielectric film between hydrophobic and hydrophilic states to shape liquid into an optical lens. Wait, there's more. Electrowetting (you're still smiling) is a subset of a mechanical device called micropumps. This begs the question: Does micropumping cause eletrowetting or relieve electrowetting? I'll let you mull over that.

— Gary A. Petersen
Shim Enterprise, Inc.
gary.petersen@shiminc.com

CrossTalk is co-sponsored by the following organizations:

Homeland Security

NAVAIR

CrossTalk / 309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737